Proposals

Open Model Context Protocol Proposals

All 42 proposals in numeric order · Built from `7596f66` · 2026-09-23

Contents

SEP-414 Document OpenTelemetry Trace Context Propagation Conventions

Final · Standards Track · Created 2025-04-25

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-04-25
- **Author(s)**: Adrian Cole (@codefromthecrypt)
- **Sponsor**: Marcelo Trylesinski (@Kludex)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/414

Abstract

This SEP documents conventions for OpenTelemetry (OTel) trace context propagation in MCP.

OTel semantic conventions for MCP specify using `_meta` as the carrier for W3C Trace Context keys. This is already in practice in the C# SDK and other implementations.

This specification documents an exception to the DNS prefixing convention for keys in `_meta`. This enables interoperability across existing and new implementations and serves as a foundation for related SEPs (such as SEP-2028).

Specification

This SEP adds documentation to the MCP specification, noting:

1. When OTel trace context is propagated via `_meta`, the keys `traceparent`, `tracestate`, and `baggage` follow W3C Trace Context and W3C Baggage value formats.
2. A non-normative example showing trace context in `_meta`.
3. A note clarifying why this an exception to DNS prefixing keys in `_meta`: to remain compatible with existing implementations and the OpenTelemetry semantic conventions.

See agentclientprotocol/agent-client-protocol#297 for equivalent documentation changes in ACP.

Non-normative example

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    },
    "_meta": {
      "traceparent": "00-0af7651916cd43dd8448eb211c80319c-00f067aa0ba902b7-01"
    }
  }
}
```

Rationale

Why document this?

This is currently documented elsewhere, but not as an MCP specification. Doing so ensures that SEPs depending on this pattern can complete, as well as other SDKs in and outside the MCP org can as well, such as Logfire and ToolHive.

If we don't document this shared concern, differing interpretations could materialize, such as namespacing traceparent like `io.modelcontextprotocol.traceparent`, which will break traces and log correlation.

Related SEPs

- SEP-1788 - reserved keys in `_meta`; should be updated with `traceparent`, `tracestate`, and `baggage` when this SEP is implemented
- SEP-2028 - builds on this SEP for forwarding `_meta` values to HTTP headers

Backward Compatibility

This SEP documents existing conventions and is backward compatible.

Security Implications

Trace context in `_meta` may include correlation IDs. Implementations should follow existing data-handling guidance appropriate to their environment.

Reference Implementation

Existing implementations using this pattern:

- C# SDK instrumentation
- Python SDK instrumentation

- OpenInference MCP instrumentation (Python)
- OpenInference MCP instrumentation (TypeScript)
- Envoy AI Gateway
- Logfire
- ToolHive

SEP-932 Model Context Protocol Governance

Final · Process · Created 2025-07-08

- **Status**: Final
- **Type**: Process
- **Created**: 2025-07-08
- **Author(s)**: David Soria Parra
- **PR**: #931
- **Issue**: #932

Abstract

This SEP establishes the formal governance model for the Model Context Protocol (MCP) project. It defines the organizational structure, decision-making processes, and contribution guidelines necessary for transparent and effective project stewardship. The proposal introduces a hierarchical governance structure with clear roles and responsibilities, along with the Specification Enhancement Proposal (SEP) process for managing protocol changes.

Motivation

As the Model Context Protocol grows in adoption and complexity, the need for formal governance becomes critical. The current informal decision-making process lacks:

1. **Transparency**: Community members have no clear visibility into how decisions are made
2. **Participation Pathways**: Contributors lack defined ways to influence project direction
3. **Accountability**: No formal structure exists for resolving disputes or contentious issues
4. **Scalability**: Ad-hoc processes cannot scale with growing community and technical complexity

Without formal governance, the project risks:

- Fragmentation of the ecosystem
- Unclear or inconsistent technical decisions
- Reduced community trust and participation
- Inability to effectively manage contributions at scale

Rationale

The proposed governance model draws inspiration from successful open source projects like Python, PyTorch, and Rust. Key design decisions include:

Hierarchical Structure

We chose a hierarchical model (Contributors → Maintainers → Core Maintainers → Lead Maintainers) that is effectively how the project decisions are made today. From there we will continue to evolve governance in the best interest of the project.

Individual vs Corporate Membership

Membership is explicitly tied to individuals rather than companies to:

- Ensure decisions prioritize protocol integrity over corporate interests
- Prevent capture by any single organization
- Maintain continuity when individuals change employers

SEP Process

The Specification Enhancement Proposal process ensures:

- All protocol changes undergo thorough review
- Community input is systematically collected
- Design decisions are documented for posterity
- Implementation precedes finalization

Specification

Governance Structure

Contributors

- Any individual who files issues, submits pull requests, or participates in discussions
- No formal membership or approval required

Maintainers

- Responsible for specific components (SDKs, documentation, etc.)
- Appointed by Core Maintainers
- Have write/admin access to their repositories
- May establish component-specific processes

Core Maintainers

- Deep understanding of MCP specification required
- Responsible for protocol evolution and project direction
- Meet bi-weekly for decisions
- Can veto maintainer decisions by majority vote
- Current members listed in governance documentation

Lead Maintainers

- Justin Spahr-Summers and David Soria Parra
- Can veto any decision
- Appoint/remove Core Maintainers
- Admin access to all infrastructure

Backwards Compatibility

N/A

Reference Implementation

See #931

1. **Documentation Files**:
   - `/docs/community/governance.mdx` - Full governance documentation
   - `/docs/community/sep-guidelines.mdx` - SEP process guidelines

Security Implications

N/A

SEP-973 Expose additional metadata for Implementations, Resources, Tools and Prompts

Final · Standards Track · Created 2025-07-15

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-15
- **Author(s)**: @jesselumarie
- **Issue**: #973

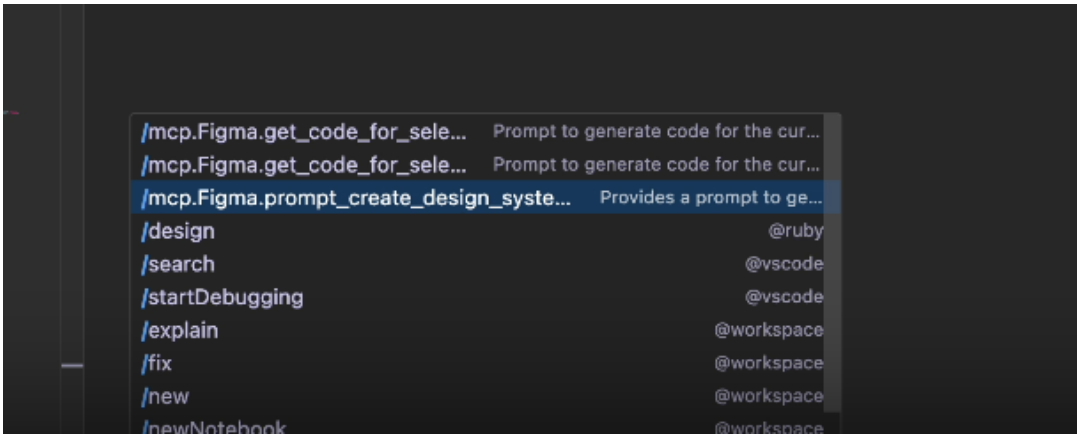
Abstract

This SEP proposes adding two optional fields— `icons` and `websiteUrl`. The `icons` and `websiteUrl` would be added to the `Implementation` schema so that clients can visually identify third-party implementations and link directly to their documentation. The `icons` parameter will also be added to the `Tool`, `Resource` and `Prompt` schemas. While this can be used by both servers and clients for all implementations, we expect it to be used initially for server-provided implementations.

Motivation

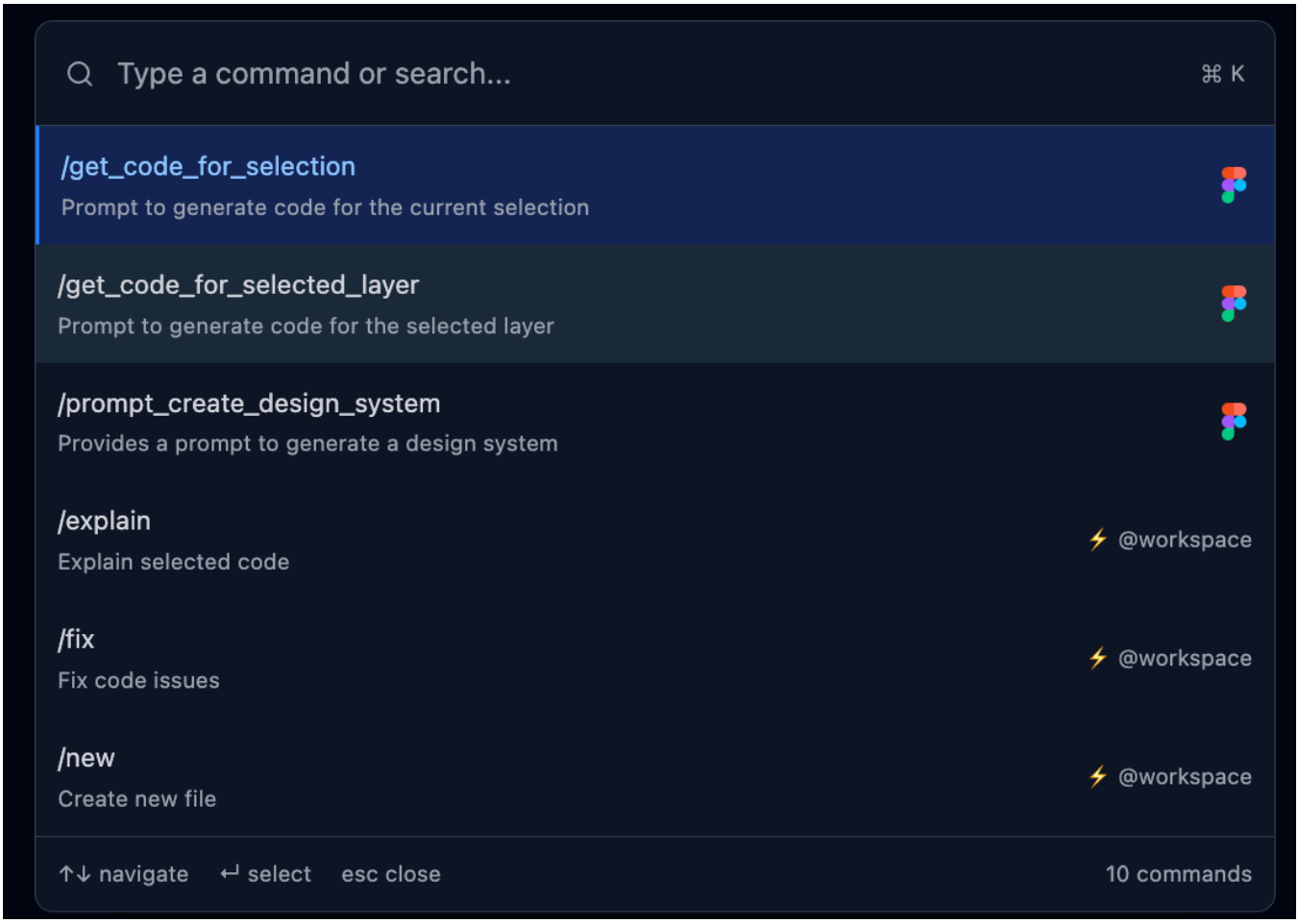
Current State

Current implementations only expose namespaced metadata, forcing clients to display generic labels with no visual cues.



Proposed State

The proposed implementation would allow us to add visual affordances and links to documentation, making it easier to visually identify which servers/clients are providing an implementation e.g. a tool in a slash command interface:



- **Visual Affordance:** Icons make it immediately clear to users which tool or resource source is in use.
- **Discoverability:** A link to documentation (`websiteUrl`) allows clients to direct users to more information with a single click.

Rationale

This design builds on prior work in web manifests (MDN) and consolidates community feedback:

- **Consolidation of PRs:** Merges the changes from PR #417 and PR #862 into a single, cohesive enhancement.
- **Flexible Icon Sizes:** Supports multiple icon sizes (e.g., `48x48` , `96x96` , or `any` for vector formats) to accommodate different client UI needs.
- **Optional Fields:** By making both fields optional, existing implementations remain fully compatible.

Specification

Extend the `Implementation` object as follows:

```
/**
 * A url pointing to an icon URL or a base64-encoded data URI
 *
 * Clients that support rendering icons MUST support at least the following MIME types:
 * - image/png - PNG images (safe, universal compatibility)
 * - image/jpeg (and image/jpg) - JPEG images (safe, universal compatibility)
 *
 * Clients that support rendering icons SHOULD also support:
 * - image/svg+xml - SVG images (scalable but requires security precautions)
 * - image/webp - WebP images (modern, efficient format)
 */
export interface Icon {
  /**
   * A standard URI pointing to an icon resource.
   *
   * Consumers MUST takes steps to ensure URLs serving icons are from the
   * same domain as the client/server or a trusted domain.
   *
   * Consumers MUST take appropriate precautions when consuming SVGs as they can contain
   * executable JavaScript
   *
   * @format uri
   */
  src: string;
  /** Optional override if the server's MIME type is missing or generic. */
  mimeType?: string;
  /** e.g. "48x48", "any" (for SVG), or "48x48 96x96" */
  sizes?: string;
}

/**
 * Describes the MCP implementation
 */
export interface Implementation extends BaseMetadata {
  version: string;
  /**
   * An optional list of icons for this implementation.
   * This can be used by clients to display the implementation in a user interface.
   * Each icon should have a `kind` property that specifies whether it is a data
representation or a URL source, a `src` property that points to the icon file or data
representation, and may also include a `mimeType` and `sizes` property.
   * The `mimeType` property should be a valid MIME type for the icon file, such as
"image/png" or "image/svg+xml".
   * The `sizes` property should be a string that specifies one or more sizes at which
the icon file can be used, such as "48x48" or "any" for scalable formats like SVG.
   * The `sizes` property is optional, and if not provided, the client should assume
that the icon can be used at any size.
   */
  icons?: Icon[];
  /**
   * An optional URL of the website for this implementation.
   *
   * Consumers MUST takes steps to ensure URLs serving icons are from the
```

```
   * same domain as the client/server or a trusted domain.
   *
   * Consumers MUST take appropriate precautions when consuming SVGs as they can contain
   * executable JavaScript
   *
   * @format: uri
   */
  websiteUrl?: string;
}
```

Extend the `Tool`, `Resource` and `Prompt` interfaces with the following type:

```
  /**
   * An optional list of icons for a resource.
   * This can be used by clients to display the resource's icon in a user interface.
   * Each icon should have a `kind` property that specifies whether it is a data
representation or a URL source, a `src` property that points to the icon file or data
representation, and may also include a `mimeType` and `sizes` property.
   * The `mimeType` property should be a valid MIME type for the icon file, such as
"image/png" or "image/svg+xml".
   * The `sizes` property should be a string that specifies one or more sizes at which
the icon file can be used, such as "48x48" or "any" for scalable formats like SVG.
   * The `sizes` property is optional, and if not provided, the client should assume
that the icon can be used at any size.
   */
  icons?: Icon[];
```

Backwards Compatibility

Both icons and websiteUrl are optional fields; clients that ignore them will fall back to existing behavior.

Security Implications

This shouldn't introduce any new security implications.

SEP-985 Align OAuth 2.0 Protected Resource Metadata with RFC 9728

Final · Standards Track · Created 2025-07-16

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-16
- **Author(s)**: sunishsheth2009
- **Issue**: #985

Abstract

This proposal brings the MCP spec's handling of OAuth 2.0 Protected Resource Metadata in line with RFC 9728.

Currently, the MCP spec requires the use of the HTTP WWW-Authenticate header when returning a 401 Unauthorized to indicate the location of the protected resource metadata. However, RFC 9728, Section 5 states:

“A protected resource MAY use the WWW-Authenticate HTTP response header field, as discussed in RFC 9110, to return a URL to its protected resource metadata to the client.”

This suggests that the MCP spec could be made more flexible while still maintaining RFC compliance.

Rationale

Many large-scale, dynamic, multi-tenant environments rely on a centralized authentication service separate from the backend resource servers. In such deployments, injecting WWW-Authenticate headers from backend services is non-trivial due to separation of concerns and infrastructure complexity.

In these scenarios, having the option to discover metadata via a well-known URL provides a practical path forward for easier MCP adoption. Requiring only the header would impose significant communication overhead between components, especially when hundreds or thousands of MCP instances are created and destroyed dynamically. Also if there are specific managed MCP servers, adopting headers across centralized system would add significant overhead.

While this increases complexity for clients—who must now implement logic to probe metadata endpoints—it reduces friction for server deployments and may encourage broader adoption. There are tradeoffs:

Pros for Server Developers: Avoid complex header injection; simplifies integration in distributed environments.

Cons for Client Developers: Clients must fall back to metadata discovery logic when the header is absent, increasing client complexity.

Proposed State

Update the MCP spec to:

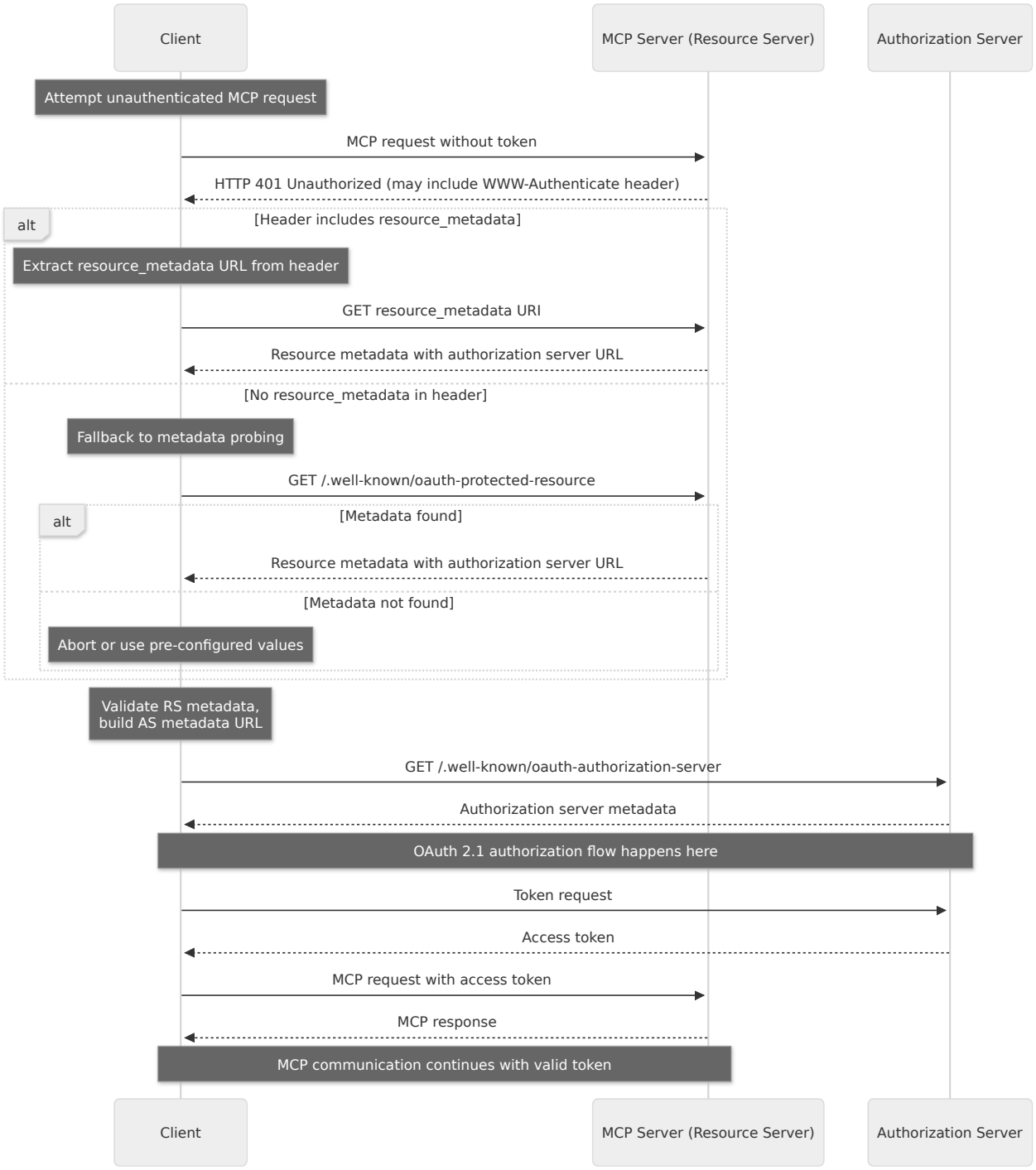
```
Clients MUST interpret the WWW-Authenticate header, and fallback to probing for metadata
if not present.
Servers SHOULD return the WWW-Authenticate header
```

The reason for deviating a bit on the RFC: Go with SHOULD over MAY for WWW-Authenticate is that it makes supporting other features, such as incremental authorization easier (e.g. you make a request for a tool, but need additional scopes, and receive a WWW-Authenticate challenge indicating the scopes).

Based on the above, following the updated flow:

- Attempt the MCP request without a token.
- If a 401 Unauthorized response is received: Check for a WWW-Authenticate header. If present and includes the resource_metadata parameter, use it to locate the resource metadata.
- If the header is absent or does not include resource_metadata, fallback to requesting /.well-known/oauth-protected-resource.

This change allows more flexible deployment models without removing existing capabilities.



Backward Compatibility

This proposal is fully backward-compatible.

It retains support for the WWW-Authenticate header (already in the spec) and introduces a fallback mechanism using the .well-known metadata path, which is already defined in MCP as a MUST-support location.

Clients that already support metadata probing benefit from improved interoperability. Servers are not required to emit the WWW-Authenticate header if it is infeasible, but doing so is still encouraged to reduce client complexity and enable future extensibility.

SEP-986 Specify Format for Tool Names

Final · Standards Track · Created 2025-07-16

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-16
- **Author(s)**: kentcdodds
- **Issue**: #986

Abstract

The Model Context Protocol (MCP) currently lacks a standardized format for tool names, resulting in inconsistencies and confusion for both implementers and users. This SEP proposes a clear, flexible standard for tool names: tool names should be 1–64 characters, case-sensitive, and may include alphanumeric characters, underscores (_), dashes (-), dots (.), and forward slashes (/). This aims to maximize compatibility, clarity, and interoperability across MCP implementations while accommodating a wide range of naming conventions.

Motivation

Without a prescribed format for tool names, MCP implementations have adopted a variety of naming conventions, including different separators, casing, and character sets. This inconsistency can lead to confusion, errors in tool invocation, and difficulties in documentation and automation. Standardizing the allowed characters and length will:

- Make tool names predictable and interoperable across clients.
- Allow for hierarchical and namespaced tool names (e.g., using / and .).
- Support both human-readable and machine-generated names.
- Avoid unnecessary restrictions that could block valid use cases.

Rationale

Community discussion highlighted the need for flexibility in tool naming. While some conventions (like lower-kebab-case) are common, many tools and clients use uppercase, underscores, dots, and slashes for namespacing or clarity. The proposed pattern—allowing a-z, A-Z, 0-9, _, -, ., and /—is based on patterns used in major clients (e.g., VS Code, Claude) and aligns with common conventions in programming and APIs. Restricting spaces and commas avoids parsing issues and ambiguity. The length limit (1–64) is generous enough for most use cases but prevents abuse.

Specification

- Tool names SHOULD be between 1 and 64 characters in length (inclusive).
- Tool names are case-sensitive.
- Allowed characters: uppercase and lowercase ASCII letters (A-Z, a-z), digits (0-9), underscore (_), dash (-), dot (.), and forward slash (/).
- Tool names SHOULD NOT contain spaces, commas, or other special characters.

- Tool names SHOULD be unique within their namespace.
- Example valid tool names:
  - getUser
  - user-profile/update
  - DATA_EXPORT_v2
  - admin.tools.list

Backwards Compatibility

This change is not backwards compatible for existing tools that use disallowed characters or exceed the new length limits. To minimize disruption:

- Existing non-conforming tool names SHOULD be supported as aliases for at least one major version, with a deprecation warning.
- Tool authors SHOULD update their documentation and code to use the new format.
- A migration guide SHOULD be provided to assist implementers in updating their tool names.

Reference Implementation

A reference implementation can be provided by updating the MCP core library to enforce the new tool name validation rules at registration time. Existing tools can be updated to provide aliases for their new conforming names, with warnings for deprecated formats. Example code and migration scripts can be included in the MCP repository.

Security Implications

None. Standardizing tool name format does not introduce new security risks.

SEP-990 Enable enterprise IdP policy controls during MCP OAuth flows

Final · Standards Track · Created 2025-06-04

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-06-04
- **Author(s)**: Aaron Parecki (@aaronpk)
- **PR**: #646
- **Issue**: #990

Abstract

This extension is designed to facilitate secure and interoperable authorization of MCP clients within corporate environments, leveraging existing enterprise identity infrastructure.

- For end users, this removes the need to manually connect and authorize the MCP Client to individual services within the organization.
- For enterprise admins, this enables visibility and control over which MCP Servers are able to be used within the organization.

How Has This Been Tested?

We have an end to end implementation of this here, and in-progress MCP implementations with some partners.

Breaking Changes

This is designed to augment the existing OAuth profile by providing an alternative when used under an enterprise IdP. MCP clients can opt in to this profile when necessary.

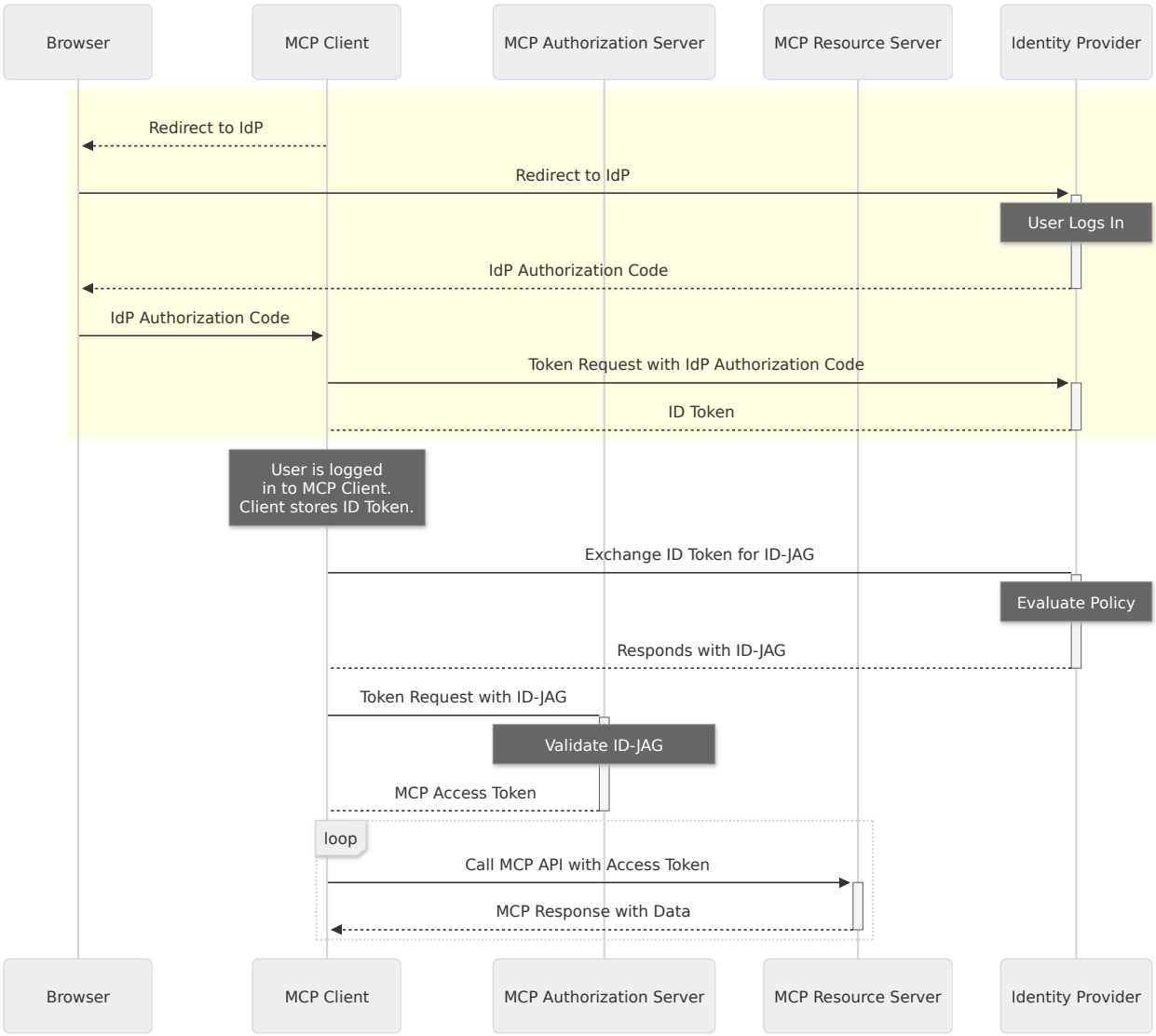
Additional Context

For more background on this problem, you can refer to my blog post about this here:

Enterprise-Ready MCP

I also presented this at the MCP Dev Summit in May.

A high level overview of the flow is below:



> [!IMPORTANT] **State:** Ready to Review

SEP-991 Enable URL-based Client Registration using OAuth Client ID Metadata Documents

Final · Standards Track · Created 2025-07-07

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-07
- **Author(s)**: Paul Carleton (@pcarleton) Aaron Parecki (@aaronpk)
- **Issue**: #991

SEP: OAuth Client ID Metadata Documents for MCP

Abstract

This SEP proposes adopting OAuth Client ID Metadata Documents as specified in draft-parecki-oauth-client-id-metadata-document-03 as an additional client registration mechanism for the Model Context Protocol (MCP). This approach allows OAuth clients to use HTTPS URLs as client identifiers, where the URL points to a JSON document containing client metadata. This specifically addresses the common MCP scenario where servers and clients have no pre-existing relationship, enabling servers to trust clients without pre-coordination while maintaining full control over access policies.

Motivation

The Model Context Protocol currently supports two client registration approaches:

1. **Pre-registration**: Requires either client developers or users to manually register clients with each server
2. **Dynamic Client Registration (DCR)**: Allows just-in-time registration by sending client metadata to a register endpoint on the Authorization server.

Both approaches have significant limitations for MCP's use case where clients frequently need to connect to servers they've never encountered before:

- Pre-registration by developers is impractical as servers may not exist when clients ship
- Pre-registration by users creates poor UX requiring manual credential management
- DCR requires servers to manage unbounded databases, handle expiration, and trust self-asserted metadata

The Target Use Case: No Pre-existing Relationship

This proposal specifically targets the common MCP scenario where:

- A user wants to connect a client to a server they've discovered
- The client developer has never heard of this server
- The server operator has never heard of this client
- Both parties need to establish trust without prior coordination

For scenarios with pre-existing relationships, pre-registration remains the optimal solution. However, MCP's value comes from its ability to connect arbitrary clients and servers, making the "no pre-existing relationship" case critical to address.

Relatedly, there are many more MCP servers than there are clients (similar to how there are many more web browsers than API's). A common scenario is an MCP server developer wanting to restrict usage to a set of clients they trust.

Key Innovation: Server-Controlled Trust Without Pre-Coordination

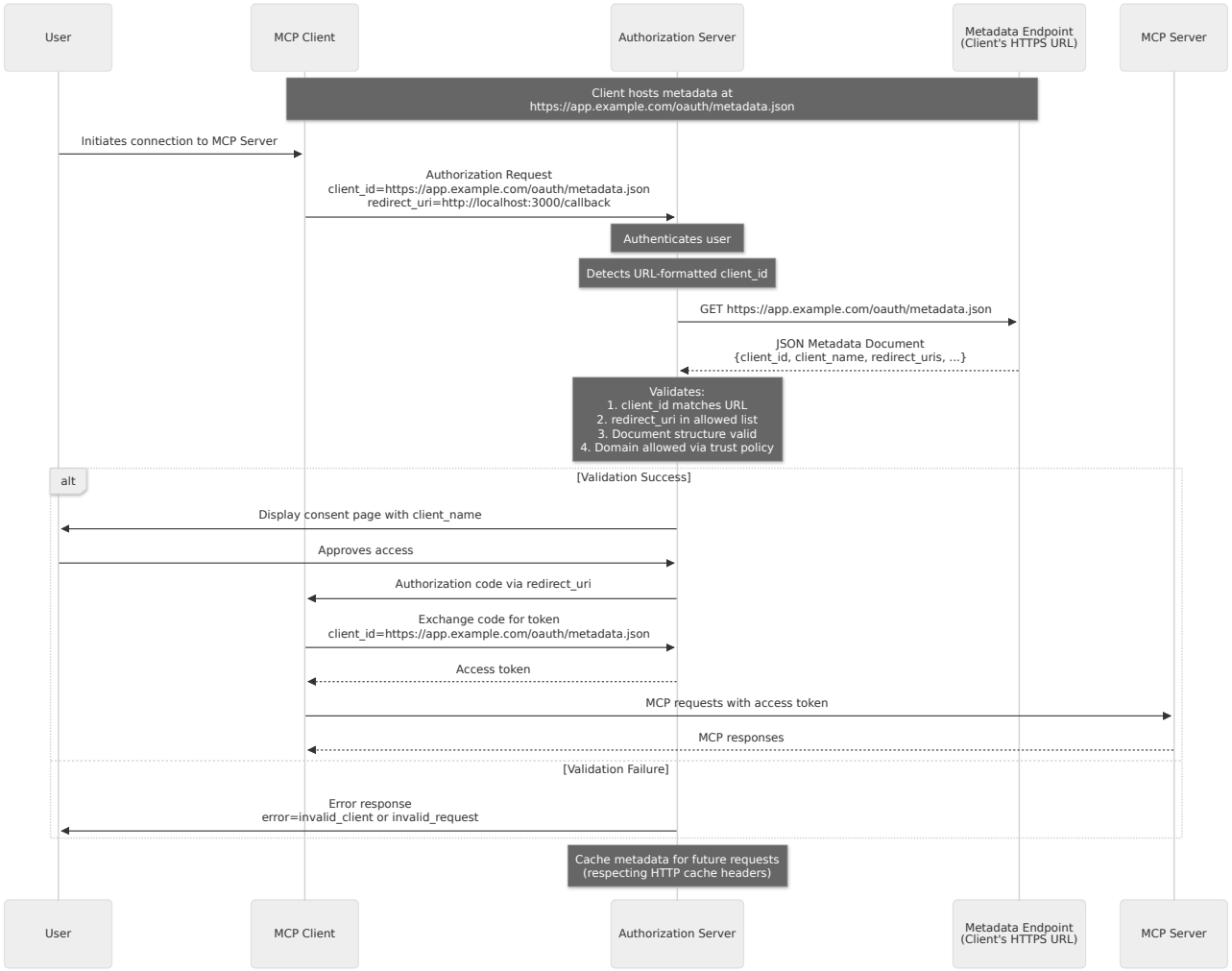
Client ID Metadata Documents enable a unique trust model where:

1. **Servers can trust clients they've never seen before** based on:
    - The HTTPS domain hosting the metadata
    - The metadata content itself
    - Domain reputation and security policies
2. **Servers maintain full control** through flexible policies:
    - **Open Servers**: Can accept any HTTPS client_id, enabling maximum interoperability
    - **Protected Servers**: Can restrict to trusted domains or specific clients
3. **No client pre-coordination required**:
    - Clients don't need to know about servers in advance
    - Clients just need to host their metadata document
    - Trust flows from the client's domain, not prior registration

Specification Changes

The change to the specification will be adding Client ID Metadata documents as a SHOULD, and changing DCR to a MAY, as we think that Client ID Metadata documents are a better default option for this scenario.

We will primarily rely on the text in the linked RFC, aiming not to repeat most of it. Below is a short version of what we'll need to specify.



Client Requirements

- Clients MUST host their metadata document at an HTTPS URL following RFC requirements
- The client_id URL MUST use "https" scheme and contain a path component
- Metadata documents MUST be valid JSON and include at minimum:
  - `client_id`: matching the document URL exactly
  - `client_name`: human-readable name for authorization prompts
  - `redirect_uris`: array of allowed redirect URIs
  - `token_endpoint_auth_method`: "none" for public clients

Note a client can use `private_key_jwt` for a `token_endpoint_auth_method` given the client metadata can provide public key information.

Server Requirements

- Servers SHOULD fetch metadata documents when encountering URL-formatted client_ids
- Servers MUST validate the fetched document contains matching client_id
- Servers SHOULD cache metadata respecting HTTP headers (max 24 hours recommended)
- Servers MUST validate redirect URIs match those in metadata document

Discovery

- Servers advertise support via OAuth metadata: `client_id_metadata_document_supported: true`

- Clients detect support and can fallback to DCR or pre-registration if unavailable

Example metadata document:

```
{
  "client_id": "https://app.example.com/oauth/client-metadata.json",
  "client_name": "Example MCP Client",
  "client_uri": "https://app.example.com",
  "logo_uri": "https://app.example.com/logo.png",
  "redirect_uris": [
    "http://127.0.0.1:3000/callback",
    "http://localhost:3000/callback"
  ],
  "grant_types": ["authorization_code"],
  "response_types": ["code"],
  "token_endpoint_auth_method": "none"
}
```

Integration with Existing MCP Auth

This proposal adds Client ID Metadata Documents as a third registration option alongside pre-registration and DCR. Servers MAY support any combination of these approaches:

- Pre-registration remains unchanged
- DCR remains unchanged
- Client ID Metadata Documents are detected by URL-formatted client_ids, and server support is advertised in OAuth metadata.

Rationale

Why This Solves the "No Pre-existing Relationship" Problem

Unlike pre-registration which requires coordination, or DCR which requires servers to manage a registration database, Client ID Metadata Documents provide:

1. **Verifiable Identity**: The HTTPS URL serves as both identifier and trust anchor
2. **No Coordination Needed**: Clients publish metadata, servers consume it
3. **Flexible Trust Policies**: Servers decide their own trust criteria without requiring client changes
4. **Stable Identifiers**: Unlike DCR's ephemeral IDs, URLs are stable and auditable

Redirect URI Attestation

A key benefit of Client ID Metadata Documents is attestation of redirect URIs:

1. **The metadata document cryptographically binds redirect URIs to the client identity** via HTTPS
2. **Servers can trust that redirect URIs in the metadata are controlled by the client** - not attacker-supplied
3. **This prevents redirect URI manipulation attacks** common with self-asserted registration

Risks of this approach

Risk: Localhost URL Impersonation

A limitation of Client ID Metadata Documents is that they cannot prevent localhost URL impersonation by itself. An attacker can claim to be any client by:

1. Providing the legitimate client's metadata URL as their client_id
2. Binding to the same localhost port the legitimate client uses
3. Intercepting the authorization code when the user approves

This attack is concerning because the server sees the correct metadata document and the user sees the correct client name, making detection difficult.

Platform-specific attestations (iOS DeviceCheck, Android Play Integrity) could address this, but they're not universally available. This would work by a developer running a backend service that consumes the DeviceCheck / Play Integrity signatures and returns a JWT usable as the `private_key_jwt` authentication for the `token_endpoint_auth_method`.

A similar approach without requiring platform-specific attestations that still raises the cost of the attack is possible using JWKS and short-lived JWTs signed by a server-side component hosted by the client developer. This component could use attestation mechanisms other than platform-specific ones to attest to the clients identity, such as the client's standard login flow. Using short lived JWTs reduces the risk of credential compromise and replay, but does not eliminate it entirely - an attacker could still proxy requests to the legitimate client's signing endpoint.

Fully mitigating this risk is outside the scope of this proposal. This proposal has the same risks as DCR does in a localhost redirect scenario.

Servers SHOULD display additional warnings for localhost-only clients.

Risk: Server Side Request Forgery (SSRF)

The authorization server takes a URL as input from an unknown client, and then fetches that URL. A malicious client could use this to send non-metadata requests on behalf of the authorization server. An example would be sending a URL corresponding to a private administration endpoint that the authorization server has access to.

This can be prevented by validating the URL's and the IP's those URL's resolve to prior to initiating a fetch request.

Risk: Distributed Denial of Service (DDoS)

Similarly, an attacker could try to leverage a pool of authorization servers to perform a denial of service attack on a non-MCP server.

There is not any additional amplification for the fetch request (i.e. the bandwidth from the client to make the request roughly equals the bandwidth of the request sent to the target server), and each authorization server can aggressively cache the result of these metadata fetches, so it is unlikely to be an attractive DDoS vector.

Risk: Maturity of referenced specification

The RFC for Client ID Metadata documents is still a draft. It has been implemented by the platform Bluesky, but has not been ratified or very widely adopted outside of that, and may evolve over time. Our intention is to evolve and align with subsequent drafts and any final standard, while minimizing disruption and breakage with existing implementations.

This approach has the risk that there are implementation challenges or flaws in the protocol that have not surfaced yet. However, even though DCR has been ratified, and it also has a number of implementation challenges that developers are facing when trying to use it in an open ecosystem context like MCP. Those challenges are the motivation behind this proposal.

Risk: Client implementation burden, especially local clients

This specification requires an additional piece of infrastructure for clients, since they need to host a metadata file behind an HTTPS url. Without this specification, a client could be strictly a desktop application for example.

The burden of hosting this endpoint is expected to be low as hosting a static JSON file is fairly straightforward and most known clients have a webpage advertising their client or providing download links.

Risk: Fragmentation of authorization approaches

Authorization for MCP is already challenging to fully implement for clients and servers. Questions about how to do it correctly and best practices are some of the most common in the community. Adding another branch to the authorization flow means this could be even more complicated and fractured, meaning fewer developers succeed in following the specification, and the promise of compatibility and an open ecosystem suffers as a result.

This proposal intends to simplify the story for authorization server and resource server developers by providing a clearer mechanism to trust redirect URIs and less operational overhead. This proposal depends on that simplicity being clearly the better option for most folks, which will drive more adoption and end up being the most supported option. If we do not believe that it is clearly the better option, then we should not adopt this proposal.

This proposal also provides a unified mechanism for both open servers and servers that want to restrict which clients can be used. Alternatives to this proposal require that clients and servers implement different mechanisms for the open and protected use cases.

Alternatives Considered

1. **Enhanced DCR with Software Statements**: More complex, requires JWKS hosting and JWT signing
2. **Mandatory Pre-registration**: Poor developer and user experience for MCP's distributed ecosystem
3. **Mutual TLS**: Requires trusting a client certificate authority, impractical in an open ecosystem
4. **Status Quo**: Continues current pain points for server implementers

Client ID Metadata document is a strict improvement over DCR for the most common open-ecosystem use case. It can be further extended in the future to better support things like OS-level attestations and jwks_uri's.

Backward Compatibility

This proposal is fully backward compatible:

- Existing pre-registered clients continue working unchanged
- Existing DCR implementations continue working unchanged
- Servers can adopt Client ID Metadata Documents incrementally
- Clients can detect support and fall back to other methods

Prototype Implementation

A prototype implementation is available here demonstrating:

1. Client-side metadata document hosting
2. Server-side metadata fetching and validation

3. Integration with existing MCP OAuth flows
4. Proper error handling and fallback behavior

Security Implications

1. **Phishing Prevention**: Display client hostname prominently
2. **SSRF Protection**: Validate URLs, limit response size, timeout requests, rate limit outbound requests

Best Practices

- Only fetch client metadata after authenticating the user
- Implement rate limiting on outbound metadata fetches
- Consider additional warnings for new/unknown/localhost domains
- Log metadata fetch failures for monitoring

References

- draft-parecki-oauth-client-id-metadata-document-03
- OAuth 2.1
- RFC 7591 - OAuth 2.0 Dynamic Client Registration
- MCP Specification - Authorization
- Evolving OAuth Client Registration in the Model Context Protocol

SEP-994 Shared Communication Practices/Guidelines

Final · Process · Created 2025-07-17

- **Status**: Final
- **Type**: Process
- **Created**: 2025-07-17
- **Author(s)**: @localden
- **Issue**: #994
- **PR**: #1002

Abstract

This SEP establishes the communication strategy and framework for the Model Context Protocol community. It defines the official channels for contributor communication, guidelines for their use, and processes for decision documentation.

Motivation

As the MCP community grows, clear communication guidelines are essential for:

- **Consistency**: Ensuring all contributors know where and how to communicate
- **Transparency**: Making project decisions visible and accessible
- **Efficiency**: Directing discussions to the most appropriate channels
- **Security**: Establishing proper processes for handling sensitive issues

Specification

Communication Channels

The MCP project uses three primary communication channels:

1. **Discord**: For real-time or ad-hoc discussions among contributors
2. **GitHub Discussions**: For structured, longer-form discussions
3. **GitHub Issues**: For actionable tasks, bug reports, and feature requests

Security-sensitive issues follow a separate process defined in SECURITY.md.

Discord Guidelines

The Discord server is designed for **MCP contributors** and is not intended for general MCP support.

Public Channels (Default)

- Open community engagement and collaborative development
- SDK and tooling development discussions
- Working and Interest Group discussions
- Community onboarding and contribution guidance
- Office hours and maintainer availability

Private Channels (Exceptions)

Private channels are reserved for:

- Security incidents (CVEs, protocol vulnerabilities)
- People matters (maintainer discussions, code of conduct)
- Coordination requiring immediate focused response

All technical and governance decisions must be documented publicly in GitHub.

GitHub Discussions

Used for structured, long-form discussion:

- Project roadmap planning
- Announcements and release communications
- Community polls and consensus-building
- Feature requests with context and rationale

GitHub Issues

Used for actionable items:

- Bug reports with reproducible steps
- Documentation improvements
- CI/CD and infrastructure issues
- Release tasks and milestone tracking

Decision Records

All MCP decisions are documented publicly:

- **Technical decisions**: GitHub Issues and SEPs
- **Specification changes**: Changelog on the MCP website
- **Process changes**: Community documentation
- **Governance decisions**: GitHub Issues and SEPs

Decision documentation includes:

- Decision makers
- Background context and motivation
- Options considered
- Rationale for chosen approach
- Implementation steps

Rationale

This framework balances openness with practicality:

- **Public by default**: Maximizes transparency and community participation
- **Private when necessary**: Protects security and personal matters
- **Channel separation**: Keeps discussions organized and searchable
- **Documentation requirements**: Ensures decisions are preserved and discoverable

Backward Compatibility

This SEP establishes new processes and does not affect existing protocol functionality.

Reference Implementation

The communication guidelines are published at:
https://openmodelcontextprotocol.org/community/communication

SEP-1024 MCP Client Security Requirements for Local Server Installation

Final · Standards Track · Created 2025-07-22

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-22
- **Author(s)**: Den Delimarsky
- **Issue**: #1024

Abstract

This SEP addresses critical security vulnerabilities in MCP client implementations that support one-click installation of local MCP servers. The current MCP specification lacks explicit security requirements for client-side installation flows, allowing malicious actors to execute arbitrary commands on user systems through crafted MCP server configurations distributed via links or social engineering.

This proposal establishes a best practice for MCP clients, requiring explicit user consent before executing any local server installation commands and complete command transparency.

Motivation

The existing MCP specification does not address client-side security concerns related to streamlined ("one-click") local server configuration. Current MCP clients that implement these configuration experiences create significant attack vectors:

1. **Silent Command Execution**: MCP clients can automatically execute embedded commands without user review or consent when installing local servers via one-click flows.
2. **Lack of Visibility**: Users have no insight into what commands are being executed on their systems, creating opportunities for data exfiltration, system compromise, and privilege escalation.
3. **Social Engineering Vulnerabilities**: Users become comfortable executing commands labeled as "MCP servers" without proper scrutiny, making them susceptible to malicious configurations.
4. **Arbitrary Code Execution**: Attackers can embed harmful commands in MCP server configurations and distribute them through legitimate channels (repositories, documentation, social media).

Visual Studio Code addressed this by implementing consent dialogs. Similarly, Cursor also supports a consent dialog for one-click local MCP server installation.

Without explicit security requirements in the specification, MCP client implementers may unknowingly create vulnerable installation flows, putting end users at risk of system compromise.

Specification

Client Security Requirements

MCP clients that support one-click local MCP server configuration **MUST** implement the following security controls:

Pre-Configuration Consent

Before executing any command to install or configure a local MCP server, the MCP client **MUST**:

1. Display a clear consent dialog that shows:
    - The exact command that will be executed, without truncation
    - All arguments and parameters
    - A clear warning that this operation may be potentially dangerous
2. Require explicit user approval through an affirmative action (button click, checkbox, etc.)
3. Provide an option for users to cancel the installation
4. Not proceed with installation if consent is denied or not provided

Rationale

Design Decisions

Mandatory Consent Dialogs: The requirement for explicit consent dialogs balances security with usability. While this adds friction to the MCP server configuration process, it prevents potential breaches from silent command execution.

Backward Compatibility

This SEP introduces new **requirements** for MCP client implementations but does not change the core MCP protocol or wire format.

Impact Assessment:

- **Low Impact**: Existing MCP servers and the core protocol remain unchanged
- **Client Implementation Required**: MCP clients must update their local server installation flows to comply with new security requirements
- **User Experience Changes**: Users will see consent dialogs where none existed before

Migration Path:

1. MCP clients can implement these changes in new versions without breaking existing functionality
2. Existing installed MCP servers continue to work normally
3. Only new installation flows require the consent mechanisms

No protocol-level backward compatibility issues exist, as this SEP addresses client behavior rather than the MCP wire protocol.

Reference Implementation

N/A

Security Implications

Security Benefits

This SEP directly addresses:

- **Arbitrary Code Execution**: Prevents silent execution of malicious commands
- **Social Engineering**: Forces users to consciously review commands before execution
- **Supply Chain Attacks**: Creates visibility into MCP server installation commands
- **Privilege Escalation**: Users can identify and reject commands requesting elevated privileges

Residual Risks

Even with these controls, risks remain:

- **User Override**: Users may approve malicious commands despite warnings
- **Sophisticated Obfuscation**: Advanced attackers may craft commands that appear legitimate
- **Implementation Gaps**: Clients may implement controls incorrectly

Risk Mitigation

These residual risks are addressed through:

- Clear warning language in consent dialogs
- Recommendation for additional security layers (sandboxing, signatures)
- Ongoing security research and community awareness

SEP-1034 Support default values for all primitive types in elicitation schemas

Final · Standards Track · Created 2025-07-22

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-22
- **Author(s)**: Tapan Chugh (chugh.tapan@gmail.com)
- **Issue**: #1034

Abstract

This SEP recommends adding support for default values to all primitive types in the MCP elicitation schema (StringSchema, NumberSchema, and EnumSchema), extending the existing support that only covers BooleanSchema.

Motivation

Elicitations in MCP offer a way to mitigate complex API designs: tools can request information on-demand rather than resorting to convoluted parameter handling. The challenge however is that users must manually enter obvious information that could be pre-populated for more natural interactions. Currently, only `BooleanSchema` supports default values in elicitation requests. This limitation prevents servers from providing sensible defaults for text inputs, numbers, and enum selections leading to more user overhead.

Real-World Example

Consider implementing an email reply function. Without elicitation, the tool becomes unwieldy:

```
def reply_to_email_thread(
    thread_id: str,
    content: str,
    recipient_list: List[str] = [],
    cc_list: List[str] = []
) -> None:
    # Ambiguity: Does empty list mean "no recipients" or "use defaults"?
    # Complex logic needed to handle different combinations
```

With elicitation, the tool signature itself can be much simpler

```python
def reply_to_email_thread(
    thread_id: str,
    content: Optional[str] = ""
) -> None:
    # Code can lookup the participants from the original thread
    # and prepare an elicitation request with the defaults setup
```

```javascript
const response = await client.request("elicitation/create", {
  message: "Configure email reply",
  requestedSchema: {
    type: "object",
    properties: {
      recipients: {
        type: "string",
        title: "Recipients",
        default: "alice@company.com, bob@company.com"  // Pre-filled
      },
      cc: {
        type: "string",
        title: "CC",
        default: "john@company.com"  // Pre-filled
      },
      content: {
        type: "string",
        title: "Message"
        default: "" // If provided in the tool above
      }
    }
  }
});
```

Implementation

A working implementation demonstrating clients require minimal changes to display defaults (~10 lines of code):

- Implementation PR: https://github.com/chughtapan/fast-agent/pull/2
- A demo with the above email reply workflow: https://asciinema.org/a/X7aQZjT2B5jVwn9dJ9sqQVkOM

Specification

Schema Changes

Extend the elicitation primitive schemas to include optional default values:

```typescript
export interface StringSchema {
  type: "string";
  title?: string;
  description?: string;
  minLength?: number;
  maxLength?: number;
  format?: "email" | "uri" | "date" | "date-time";
  default?: string; // NEW
}

export interface NumberSchema {
  type: "number" | "integer";
  title?: string;
  description?: string;
  minimum?: number;
  maximum?: number;
  default?: number; // NEW
}

export interface EnumSchema {
  type: "string";
  title?: string;
  description?: string;
  enum: string[];
  enumNames?: string[];
  default?: string; // NEW - must be one of enum values
}

// BooleanSchema already has default?: boolean
```

Behavior

1. The `default` field is optional, maintaining full backward compatibility
2. Default values must match the schema type
3. For EnumSchema, the default must be one of the valid enum values
4. Clients that support defaults SHOULD pre-populate form fields. Clients that don't support defaults MAY ignore the field entirely.

Rationale

1. The high-level rationale is to follow the precedent set by BooleanSchema rather than creating new mechanisms.
2. Making defaults optional ensures backward compatibility.
3. This maintains the high-level intuition of keeping the client implementation simple.

Alternatives Considered

1. **Server-side Templates**: Servers could maintain templates separately, but this adds complexity
2. **New Request Type**: A separate request type for forms with defaults would fragment the API
3. **Required Defaults**: Making defaults required would break existing implementations

Backwards Compatibility

This change is fully backward compatible with no breaking changes. Clients that don't understand defaults will ignore them, and existing elicitation requests continue to work unchanged. Clients can adopt default support at their own pace

Security Implications

No new security concerns:

1. **No Sensitive Data**: The existing guidance against requesting sensitive information still applies
2. **Client Control**: Clients retain full control over what data is sent to servers
3. **User Visibility**: Default values are visible to users who can modify them before submission

SEP-1036 URL Mode Elicitation for secure out-of-band interactions

Final · Standards Track · Created 2025-07-22

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-22
- **Author(s)**: Nate Barbettini (@nbarbettini) and Wils Dawson (@wdawson)
- **Issue**: #1036

Abstract

This SEP introduces a new `url` mode for the existing elicitation client capability, enabling secure out-of-band interactions that bypass the MCP client. URL mode elicitation addresses sensitive use cases that form mode elicitation cannot, such as gathering sensitive credentials, performing OAuth flows for external (3rd-party) authorization, and handling payments, *without* exposing sensitive data to the MCP client. By directing users to trusted URLs in their browser, this mode maintains security boundaries while enabling rich integrations with third-party services.

Motivation

The current MCP specification (2025-06-18) provides an elicitation mechanism for gathering non-sensitive information from users through structured, in-band requests (most commonly imagined as the MCP client rendering a form to collect data from the end-user). However, several critical use cases require interactions that must not pass through the MCP client:

1. Sensitive data collection: API keys, passwords, and other credentials must never transit through intermediary systems.
2. External authorization: MCP servers often need to access third-party APIs on behalf of users. The MCP authorization specification only covers client-to-server authorization, not server-to-third-party authorization. The Security Best Practices document explicitly forbids token passthrough, requiring a secure mechanism for external (3rd-party) OAuth flows. This was a particularly important motivating factor emerging from discussions in #234 and #284.
3. Payment and Subscription Flows: Financial transactions require PCI compliance and secure payment processing that cannot be achieved through in-band data collection.

Without a standardized mechanism for these interactions, MCP servers must resort to non-standard workarounds or insecure practices like requesting API keys through in-band, form-style elicitation. This SEP addresses these gaps by introducing a URL elicitation mode that leverages established web security patterns to handle sensitive interactions securely.

URL elicitation is fundamentally different from MCP authorization. URL elicitation is not for authorizing the MCP client's access to the MCP server (that's handled directly by MCP authorization). Instead, it's used when the MCP server needs to obtain sensitive information or third-party authorization on behalf of the user. The MCP client's bearer token remains unchanged, and the client's only responsibility is to provide the user with context about the elicitation URL the server wants them to open.

Specification

Overview

Elicitation is updated to support two modes:

- **Form mode** (in-band): Servers can request structured data from users with optional JSON schemas to validate responses (no change here, other than adding a name to the existing capability)
- **URL mode** (out-of-band): Servers can direct users to external URLs for sensitive interactions that must not pass through the MCP client

Capabilities

Clients that support elicitation **MUST** declare the `elicitation` capability during initialization:

```
{
  "capabilities": {
    "elicitation": {
      "form": {},
      "url": {}
    }
  }
}
```

For backwards compatibility, an empty capabilities object is equivalent to declaring support for `form` mode only:

```
{
  "capabilities": {
    "elicitation": {},
  },
}
```

Clients declaring the `elicitation` capability **MUST** support at least one mode (`form` or `url`).

Form Elicitation Requests

The only change from the existing specification is the addition of a `mode` field in the `elicitation/create` request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "elicitation/create",
  "params": {
    "mode": "form", // New field
    "message": "Please provide your GitHub username",
    "requestedSchema": {
      "type": "object",
      "properties": {
        "name": {
          "type": "string"
        }
      },
      "required": ["name"]
    }
  }
}
```

URL Elicitation Requests

URL elicitation requests **MUST** specify `mode: "url"` and include these parameters:

Name	Type	Description
`url`	string	The URL that the user should navigate to.
`elicitationId`	string	A unique identifier for the elicitation.
`message`	string	A human-readable message explaining why the interaction is needed.

Example: OAuth Authorization Flow

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "elicitation/create",
  "params": {
    "mode": "url",
    "elicitationId": "550e8400-e29b-41d4-a716-446655440000",
    "url": "https://github.com/login/oauth/authorize?
client_id=abc123&state=xyz789&scope=repo",
    "message": "Please authorize access to your GitHub repositories to continue."
  }
}
```

Response Actions

URL elicitation responses use the same three-action model as form elicitation:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "action": "accept" // or "decline" or "cancel"
  }
}
```

The response with `action: "accept"` indicates that the user has consented to the interaction. The interaction occurs out of band and the client is not aware of the outcome unless the server sends a completion notification.

Completion Notifications

Servers **SHOULD** send a `notifications/elicitation/complete` notification when an out-of-band interaction started by URL mode elicitation is completed. This allows clients to react programmatically if appropriate.

- The notification **MUST** only be sent to the client that initiated the elicitation request.
- The notification **MUST** include the `elicitationId` established in the original `elicitation/create` request.
- Clients **MUST** ignore notifications referencing unknown or already-completed IDs.
- If a completion notification never arrives, clients **SHOULD** provide a manual way for the user to continue the interaction.

Clients **MAY** use the notification to automatically retry requests that received a URL elicitation required error, update the user interface, or otherwise continue an interaction. However, because delivery of the notification is not guaranteed, clients must not wait indefinitely for a notification from the server.

```
{
  "jsonrpc": "2.0",
  "method": "notifications/elicitation/complete",
  "params": {
    "elicitationId": "550e8400-e29b-41d4-a716-446655440000"
  }
}
```

URL Elicitation Required Error

When a request cannot be processed until an elicitation is completed, the server **MAY** return a `URLElicitationRequiredError` (code `-32042`) to indicate that a URL mode elicitation is required. The server **MUST NOT** return this error except when URL mode elicitation is required by the user interaction.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "error": {
    "code": -32042,
    "message": "This request requires more information.",
    "data": {
      "elicitations": [
        {
          "mode": "url",
          "elicitationId": "550e8400-e29b-41d4-a716-446655440000",
          "url": "https://oauth.example.com/authorize?
client_id=abc123&response_type=code&...",
          "message": "Authorization is required to access your Example Co files."
        }
      ]
    }
  }
}
```

Any elicitations returned in the error **MUST** be URL mode elicitations and include an `elicitationId`.

Returning a `URLElicitationRequiredError` is equivalent to sending an `elicitation/create` request. The server may return an error (instead of sending a separate `elicitation/create` request) as an affordance to the client to make it clear that a particular elicitation is directly related to a failed client request.

The client must treat `URLElicitationRequiredError` responses as equivalent to `elicitation/create` requests. Clients may automatically retry the failed request after the elicitation is completed successfully, for example after receiving a completion notification.

Rationale

Design Decisions

Why extend elicitation instead of creating a new mechanism?

Initially, we considered creating a separate mechanism for out-of-band interactions (discussed in #475). However, after discussions with the MCP maintainers, we decided to extend the existing elicitation specification because:

1. Both mechanisms serve the same fundamental purpose: gathering information from users
2. Having two similar-but-separate mechanisms for the same purpose is confusing and error-prone
3. The `mode` parameter cleanly separates the two interaction patterns

Why can't the client perform the interaction itself?

It is tempting to suggest that the MCP client should perform the interaction itself, e.g. act as an OAuth client to a third-party authorization server. However, there are several reasons why this is not a good idea:

- If the MCP client obtains user tokens from a third-party authorization server, the MCP server becomes a token passthrough server, which is explicitly forbidden.
- Similarly, for payment-type flows, the MCP client would need to perform PCI-compliant payment processing, which is not a desired requirement for MCP clients.

Why doesn't the server block (wait) on the elicitation to complete?

URL mode elicitation requests are asynchronous or "disconnected" flows by design, because the kinds of interactions they enable are inherently asynchronous. Payment flows, external authorization, etc. can take minutes or more to complete, and in some cases never complete at all (if abandoned by the end-user).

Why disallow URLs in form mode?

Being very explicit about when URLs can (and cannot) be sent in an elicitation request improves the client's security posture. By clearly stating in the spec that URLs are *only* allowed in the `url` field of a URL mode elicitation request, client implementers can implement UX patterns that are consistent with the security model. For example, a client could refuse to render a URL as a clickable hyperlink in a form mode elicitation request, reducing the likelihood of a user clicking on a malicious URL sent by a malicious server.

Alternative Approaches Considered

1. **Token Passthrough**: Simply passing the MCP client's token to external services was rejected due to security concerns documented in the Security Best Practices. Having the MCP client obtain additional tokens and passing those to the MCP server was rejected for the same reason.

2. **OAuth-specific Capability**: Creating a capability specific to external (3rd-party) authorization with OAuth was considered, but rejected in favor of the more general URL mode elicitation approach that supports multiple use cases.

Community Feedback

This proposal incorporates extensive community feedback from discussions in #475, #234, and #284, as well as the #auth-wg working group on Discord. The community identified the need for:

- Secure credential collection without client exposure
- External authorization patterns separate from MCP authorization
- Payment and subscription flow support
- Clear security boundaries and trust models

Backward Compatibility

This SEP introduces the following breaking changes:

1. **Capability Declaration**: Clients must now specify which elicitation modes they support:

   ```
   {
     "capabilities": {
       "elicitation": {
         "form": {},
         "url": {}
       }
     }
   }
   ```

   Previously, clients only declared `"elicitation": {}` without mode specification.

2. **Mode Parameter**: All `elicitation/create` requests must now include a `mode` parameter (`"form"` or `"url"`).

Migration Path

To ease migration:

- Servers SHOULD check client capabilities before sending mode-specific requests
- Clients MAY initially support only form mode to maintain compatibility
- Existing form elicitation implementations continue to work with the addition of the mode parameter

Reference Implementation

Client/server implementation in TypeScript: feat/url-elicitation

Explainer video: https://drive.google.com/file/d/1llCFS9wmkK_RUgi5B-zHfUUgy-CNb0n0/view?usp=sharing

Security Implications

This SEP introduces several security considerations:

URL Security Requirements

1. **SSRF Prevention**: Clients must validate URLs to prevent Server-Side Request Forgery attacks
2. **Protocol Restrictions**: Only HTTPS URLs are allowed for URL elicitation
3. **Domain Validation**: Clients must clearly display target domains to users

Trust Boundaries

URL elicitation explicitly creates clear trust boundaries:

- The MCP client never sees sensitive data obtained by the MCP server via URL elicitation
- The MCP server must independently verify user identity
- Third-party services interact directly with users through secure browser contexts

Identity Verification

Servers must verify that the user completing a URL elicitation is the same user who initiated the request. Verifying the identity of the user must not rely on untrusted input (e.g. user input) from the client.

Implementation Requirements

1. **Clients must**:
    - Use secure browser contexts that prevent inspection of user inputs
    - Validate URLs for SSRF protection
    - Obtain explicit user consent before opening URLs
    - Clearly display target domains
2. **Servers must**:
    - Bind elicitation state to authenticated user sessions
    - Verify user identity at the beginning and end of a URL elicitation flow
    - Implement appropriate rate limiting
3. **Both parties should**:

    - Log security events for audit purposes
    - Implement timeout mechanisms for elicitation requests
    - Provide clear error messages for security failures

Relationship to Existing Security Measures

This proposal builds upon and complements existing MCP security measures:

- Works within the existing MCP authorization framework (MCP authorization is not affected by this proposal)
- Follows Security Best Practices regarding token handling
- Maintains separation of concerns between client-server and server-third-party authorization

SEP-1046 Support OAuth client credentials flow in authorization

Final · Standards Track · Created 2025-07-23

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-07-23
- **Author(s)**: Darin McAdams (@D-McAdams)
- **Issue**: #1046

Preamble

Title: Support OAuth client credentials flow in authorization Author: Darin McAdams (@D-McAdams) Status: Proposal Type: Standards Track Created: 2025-07-23

Abstract

Recommends adding the OAuth client credentials flow to the authorization spec to enable machine-to-machine scenarios.

Motivation

The original authorization spec mentioned the client credentials flow, but it was dropped in subsequent revisions. Therefore, the spec is currently silent on how to solve machine-to-machine scenarios where an end-user is unavailable for interactive authorization.

Specification

The authorization spec would be amended to list the OAuth client credentials flow as being allowed. Adhering to the patterns established by OAuth 2.1, the specification would RECOMMEND the use of asymmetric methods defined in RFC 753 (JWT Assertions), but also allow client secrets.

As guidance to implementors, the spec overview would also be updated to describe the different flows and when each is applicable. In addition, to address a common question, the spec would be updated to indicate that implementors may implement other authorization scenarios beyond what's defined; emphasizing that the specification defines the baseline requirements.

Rationale

To maximize interoperability (and minimize SDK complexity), this change would intentionally constrain the client credentials flow to two options:

1. JWT Assertions as per RFC 7523 (RECOMMENDED)
2. Client Secrets via HTTP Basic authentication (Allowed for maximum compatibility with existing systems)

Other options, such as mTLS, are not included.

While the spec encourages the use of RFC 7523 (JWT Assertions), it does not yet specify how to populate the JWT contents nor how to discover the client's JWKS URI to validate the JWT. In future iterations of the spec, it will be beneficial to do so. However, this was currently left unspecified pending maturity of other RFCs that can define these profiles. The other RFCs include WIMSE Headless JWT Authentication (for specifying JWT contents) and Client ID Metadata (for specifying the JWKS URI). This revision intentionally leaves extensibility for these future profiles. As a practical matter, this means implementers needing to ship solutions ASAP will most likely use client secrets which are widely supported today, whereas the JWT Assertion pattern represents the longer-term direction.

Backward Compatibility

This change is fully backward compatible. It introduces a new authorization flow, but does not alter the existing flows.

Security Implications

The specification refers to the existing OAuth security guidance.

SEP-1302 Formalize Working Groups and Interest Groups in MCP Governance

Final · Standards Track · Created 2025-08-05

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-08-05
- **Author(s)**: tadasant
- **Issue**: #1302

PR: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1350

Abstract

A short (~200 word) description of the technical issue being addressed.

In SEP-994, we introduced a notion of “Working Groups” and “Interest Groups” that facilitate MCP sub-communities for discussion and collaboration. This SEP aims to formally define those two terms: what they are meant to achieve, how groups can be created, how they are governed, and how they can be retired.

Interest Groups work to define *problems* that MCP should solve by facilitating *discussions*, while Working Groups push forward specific *solutions* by collaboratively producing *deliverables* (in the form of SEPs or community-owned implementations of the specification). Interest Group input is a welcome (but not required) justification for creation of a Working Group. Interest Group or Working Group input is collectively a welcome (but not required) input into a SEP.

Motivation

The motivation should clearly explain why the existing protocol specification is inadequate to address the problem that the SEP solves.

The community has already been self-organizing into several disparate systems for these collaborative groups:

- The Steering group has had a long-standing practice of managing a handful of collaborative groups through Discord channels (e.g. security, auth, agents). See bottom of MAINTAINERS.md.
- The “CWG Discord” has had a semi-formal process for pushing equivalent grassroots initiatives, mostly in pursuit of creating artifacts for SEP consideration (e.g. hosting, UI, tool-interfaces, search-tools)

With SEP-994 resulting in the merging of the Discord communities, we have a need to:

- Merge the existing initiatives into one unified approach, so when we reference “working group” or “interest group”, everyone knows what that means and what kind of weight the reference might carry
- Standardize a process around the creation (and eventual retirement) of such groups
- Properly distinguish between “working” and “interest” groups; the CWG experience has shown two very different motivations for starting a group worth treating with different expectations and lifecycle. Put succinctly, “interest” groups are about brainstorming possible *problems*, and “working” groups are about pushing forward specific *solutions*.

These groups exist to:

- **Facilitate high signal spaces for discussion** such that those opting into notifications and meetings feel most content is relevant to them and they can meaningfully contribute their experience and learn from others
- **Create norms, expectations, and single points of involved leadership** around making collaborative progress towards concrete deliverables that help evolve MCP

It will also form the foundation for cross-group initiatives, such as maintaining a calendar of live meetings.

Specification

The technical specification should describe the syntax and semantics of any new protocol feature. The specification should be detailed enough to allow competing, interoperable implementations. A PR with the changes to the specification should be provided.

Interest Groups (IG) [Problems]

Goal: facilitate discussion and knowledge-sharing among MCP community members with similar interests surrounding some MCP sub-topic or context. The focus is on collecting *problems* that may or may not be worth solving with SEPs or other community artifacts.

Expectations:

- At least one substantive thread / conversation per month
- AND/OR a live meeting attended by 3+ unaffiliated individuals

Examples:

- Security in MCP (currently: #security)
- Auth in MCP (currently: #auth)
- Using MCP in an internal enterprise setting (currently: #enterprise-wg)
- Tooling and practices surrounding hosting MCP servers (currently: #hosting-wg)
- Tooling and practices surrounding implementing MCP clients (currently: #client-implementors)

Lifecycle:

- Creation begins by filling out a template in #wg-ig-group-creation Discord channel
- A community moderator will review and call for a vote in the (private) #community-moderators Discord channel. Majority positive vote by members over a 72h period approves creation of the group. Can be reversed at any time (e.g. after more input comes in). Core and lead maintainers can veto.
- Facilitator(s) and Maintainer(s) responsible for organizing IG into meeting expectations
    - Facilitator is an informal role responsible for shepherding or speaking for a group
    - Maintainer is an official representative from the MCP steering group (not required for every group to have this)
- IG is retired only when community moderators or core+ maintainers decide it is not meeting expectations
    - This means successful IG’s will live on in perpetuity

Creation Template:

- Facilitator(s)
- Maintainer(s) (optional)
- Flag potential overlap with other IG’s
- How this IG differentiates itself from the related IG’s
- First topic you want to discuss

There is no requirement to be part of an IG to start a WG, or even to start a SEP. However, forming consensus in IG’s to support justifying the creation of a WG is often a good idea. Similarly, citing IG or WG support of a SEP helps the SEP as well.

Working Groups (WG) [Solutions]

Goal: facilitate MCP community collaboration on a specific SEP, themed series of SEPs, or officially endorsed Project.

Expectations:

- Minimum monthly progress towards at least one SEP or spec-related implementation OR holds maintenance responsibilities for a Project
- Facilitator(s) is/are responsible for fielding status update requests by community moderators or maintainers

Examples:

- Registry
- Inspector
- Tool Filtering
- Server Identity

Lifecycle:

- Creation begins by filling out a template in #wg-ig-group-creation Discord channel
- A community moderator will review and call for a vote in the (private) #community-moderators Discord channel. Majority positive vote by members over a 72h period approves creation of the group. Can be reversed at any time (e.g. after more input comes in). Core and lead maintainers can veto.
- Facilitator(s) and Maintainer(s) responsible for organizing WG into meeting expectations
    - Facilitator is an informal role responsible for shepherding or speaking for a group
    - Maintainer is an official representative from the MCP steering group (not required for every group to have this)
- WG is retired when either:
    - Community moderators or core+ maintainers decide it is not meeting expectations
    - The WG does not have a WIP Issue/PR for at least a month, or has completed all Issues/PRs it intends to pursue.

Creation Template:

- Facilitator(s)
- Maintainer(s) (optional)
- Explanation of interest/use cases (ideally from an IG but can come from anywhere)
- First Issue/PR/SEP you intend to procure

WG/IG Facilitators

A “Facilitator” role in a WG or IG does *not* result in a maintainership role across the MCP organization. It is an informal role into which anyone can self-nominate, responsible for helping shepherd discussions and collaboration within the group.

Core Maintainers reserve the right to modify the list of Facilitators and Maintainers for any WG/IG at any time.

PR for the changes to our documentation we'd want to enact this SEP:
https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1350

Rationale

The rationale explains why particular design decisions were made. It should describe alternate designs that were considered and related work. The rationale should provide evidence of consensus within the community and discuss important objections or concerns raised during discussion.

The design above comes from experience in facilitating the creation of + observing the behavior of informal “Community Working Groups” in the CWG Discord, and leading one of / participating in / observing the “Steering Committee Working Groups”. While the Steering WG’s were usually informally created by Lead Maintainers, the CWG Discord had a lightweight WG-creation process that involved similar steps to the proposal above (community members would propose WG’s in #working-group-ideation, and moderators would create channels from that collaboration).

As precedent, the WG and IG concepts here are similar to W3C’s notion of Working Groups and Interest Groups.

Considerations

In proposing the WG/IG design, we took the following into consideration:

Clear on-ramp for community involvement

A very common question for folks looking to invest in the MCP ecosystem is, "how do I get involved?"

These IG and WG abstractions help provide an elegant on-ramp:

1. Join the Discord, follow the conversation in IGs relevant to you. Attend live calls. Participate.
2. Offer to facilitate calls. Contribute your use cases in SEP proposals and other work.
3. When you're comfortable contributing to deliverables, jump in to contribute to WG work.
4. Do this for a period of time, get noticed by WG maintainers to get nominated as a new maintainer.

Minimal changes to existing governance structure

We did not want this change to introduce new elections, appointments, or other notions of leadership. We leverage community moderators to thumbs-up creation of new groups, allow core maintainers to veto, maintainership status stays unchanged, and the notion of "facilitator" is new but self-nominated, so does not introduce any new governance processes.

Alignment with current status quo

There is a clear "migration" path for the existing "CWG" working groups and Steering working groups - just a matter of sorting out what is "working" vs. "interest", but functionally this proposal stays out of the way of changing anything that has been working within each group's existing structure.

Nature of requests for gathering spaces

It has been clear from the requests to CWG that some groups form with a motivation to collaborate on some deliverable (e.g. `search-tools`), and others form due to common interests and a want for sub-community but not yet specific deliverables (e.g. `enterprise`). Hence, we separate the motivations into Working Groups vs. Interest Groups.

Potential for overlap in scope

In the requests for new group spaces, it is sometimes non-obvious why a new one needs to exist. For example, the stated motivation for `enterprise` at times sounded like it may just be another flavor of `hosting`. We ultimately settled on a distinction that made it clear one was not a direct subset of the other, but the concern of making clear boundaries between groups (and letting community moderators / maintainers centralize the decision-making around "what are the right layers of abstraction") is what led to the questions in the creation templates around e.g. "flag potential overlap with other IG’s".

Path to retiring stale groups

Many working groups in the old CWG and Steering models have gone stale since creation. They serve no real purpose and should be retired. For this, we introduce the formal concept of facilitators and optional maintainers in groups; and the community moderator right to retire them. By having at least informal leadership in place per group, a moderator can easily make the decision to retire a group if everyone is in agreement to proceed.

Alternatives Considered

Hierarchy between IGs and WGs

We considered *requiring* that WGs be owned or spawned by a "sponsor" IG, for the purpose of more clearly exhibiting a progression of ideas to the community; but decided against this requiring to avoid adding a new layer of governance and alignment with how the less formal groups works today.

A single WG concept (instead of both WG and IG)

There has been regular tension in both CWG and the Steering group around the question of "is XYZ really a working group? how will maintainership work?" By making IG's explicitly discussion-oriented and maintainership involvement optional, we create a space to drive those discussions without requiring some formal expectation of deliverables like we might in a well-defined WG.

Free-for-all WG/IG creation process

While very community-driven, the concern of group overlap would quickly fragment the conversations and collaboration to an untenable level; we need a centralized point of discernment here.

Backward Compatibility

All SEPs that introduce backward incompatibilities must include a section describing these incompatibilities and their severity. The SEP must explain how the author proposes to deal with these incompatibilities.

There is no major change suggested in the day to day of existing groups - the expectations laid out of IGs and WGs are easily met by existing active groups as long as they keep doing as they are doing.

A migration path for all groups is laid out below.

Reference Implementation

The reference implementation must be completed before any SEP is given status “Final”, but it need not be completed before the SEP is accepted. While there is merit to the approach of reaching consensus on the specification and rationale before writing code, the principle of “rough consensus and running code” is still useful when it comes to resolving many discussions of protocol details.

The below is the suggested migration path for each group. "Migration" just involves acknowledgement of this SEP and the expectations of each group, plus methodology for possible eventual retirement (or immediate retirement, in some cases).

After this SEP is approved, we can ping each of the groups to confirm they are on board with the migration plan.

Steering Working Groups

- All official SDK groups --> Working Groups

- Registry --> Working Group
- Documentation --> Working Group
- Inspector --> Working Group
- Auth --> Interest Group + some WGs: client-registration, improve-devx, profiles, tool-scopes
- Agents --> Working Group [Long Running / Async Tool Calls; unless we want an Agents IG on top of that?]
- Connection Lifetime --> Retire
- Streaming --> Retire
- Spec Compliance --> Retire (good idea but stale; would be good for someone to spearhead a new Working Group)
- Security --> Interest Group (perhaps with Security Best Practices WG?)
- Transports --> Interest Group
- Server Identity --> Working Group
- Governance --> Working Group (or Retire if no more work here?)

Community Working Groups

- agent-comms --> Retire
- enterprise --> Interest Group (request a proposal to start)
- hosting --> Interest Group (request a proposal to start)
- load-balancing --> Retire
- model-awareness --> Working Group (request a proposal to start)
- search-tools (tool-filtering) --> Working Group
- server-identity --> merge with Steering equivalent
- security --> merge with Steering equivalent
- server-identity --> merge with Steering equivalent
- tool-interfaces --> Retire
- ui --> Interest Group
- schema-validation --> Retire (same as Steering equivalent)

SEP-1303 Input Validation Errors as Tool Execution Errors

Final · Standards Track · Created 2025-08-05

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-08-05
- **Author(s)**: @fredericbarthelet
- **Issue**: #1303

Abstract

This SEP proposes treating tools input validation errors as Tool Execution Errors rather than Protocol Errors. This change would enable language models to receive validation error feedback in their context window, allowing them to self-correct and successfully complete tasks without human intervention, significantly improving task completion rate.

Motivation

Language models can learn from tool input validation error messages and retry a tools/call with corrected parameters accordingly, but only if they receive the error feedback in their context window. Protocol Errors are catch at the application level by the MCP Client. Only Tool Execution Errors are forwarded back to the model as JSON-RPC responses. With the current specifications, models cannot see these error messages and thus cannot self-correct, leading to repeated failures and poor user experiences.

Problem Statement

Consider a flight booking tool that validates departure dates using the following `zod` validation schema:

```
departureDate: z.string()
  .regex(/^\d{2}\/\d{2}\/\d{4}$/, "date must be in dd/mm/yyyy format")
  .superRefine((dateStr, ctx) => {
    const date = parseDateFr(dateStr);
    if (date.getTime() < Date.now()) {
      ctx.addIssue({
        code: z.ZodIssueCode.custom,
        message:
          "Dates must be in the future. Current date is " +
          formatDateFr(new Date()),
      });
    }
    return true;
  })
  .describe("Departure date in dd/mm/yyyy format");
```

Tool expected input JSON schema can only describe the regex statement. The actual programmatic check that the date is in the past cannot be expressed here as JSON schema. Even when a model provides a syntactically correct date that passes JSON schema validation, there is no guarantee it will be in the future. When a validation error is raised and returned as a Protocol Error:

1. The model doesn't receive the error message explaining why the date was rejected
2. The model repeats the same mistake multiple times (e.g., Cursor typically consistently sends dates in 2024 when the user only specify day and month or relative date and repeats the same tools/call request 3 times without getting any information as to why the tools call fails)
3. The task fails despite the model being capable of correcting itself if given proper feedback
4. Users experience frustration and must manually intervene

Benefits of This Proposal

1. **Higher Task Completion Rates**: Models can self-correct validation errors without human intervention
2. **Better User Experience**: Reduced failures and faster task completion
3. **Leverages Model Capabilities**: Modern LLMs excel at understanding and responding to error messages
4. **Reduced API Calls**: Fewer retry attempts as models correct themselves on the first error

Specification

Current Behavior

The tool errors specification currently provides ambiguous guidance:

- "Invalid arguments" should be treated as Protocol Error
- "Invalid input data" should be treated as Tool Execution Error

This ambiguity leads to inconsistent implementations where valuable error feedback is lost.

Proposed Change

Clarify the specification with the following changes:

1. Removes the "invalid argument" category from **Protocol Errors**.
2. **Tool Execution Errors** should be used for all tool argument validation failures (merging `invalid argument` and `invalid input data` under a new `input validation errors` category)

Specification Text Changes

Update the error handling section to include:

```
## Error Handling

Tools use two error reporting mechanisms:

1. **Protocol Errors**: Standard JSON-RPC errors for issues like:

   - Unknown tools
   - Server errors

2. **Tool Execution Errors**: Reported in tool results with `isError: true`:
   - API failures
   - Input validation errors
   - Business logic errors
```

Implementation

Before (Protocol Error)

```
// Model submits past date
request: {
  ...
  method: "tools/call",
  params: {
    name: "book_flight",
    arguments: {
      departureDate: "12/12/2024"  // Past date
    }
  }
}

// Server returns Protocol Error
response: {
  ...
  error: {
    code: -32602,
    message: "Invalid params"
  }
}

// Model retries blindly with another past date
// This cycle repeats until failure
```

After (Tool Execution Error)

```
// Model submits past date
request: {
  ...
  method: "tools/call",
  params: {
    name: "book_flight",
    arguments: {
      departureDate: "12/12/2024"  // Past date
    }
  }
}

// Server returns Tool Execution Error (visible to model)
response: {
  ...
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Dates must be in the future. Current date is 08/08/2025"
      }
    ],
    "isError": true
  }
}

// Model understands the error and corrects itself
request: {
  method: "tools/call",
  params: {
    name: "book_flight",
    arguments: {
      departureDate: "12/12/2025"  // Future date
    }
  }
}
```

Backwards Compatibility

This change is backwards compatible as it:

- Does not alter the protocol structure
- Only clarifies existing ambiguous behavior
- Maintains all existing error types and formats
- Improves behavior without breaking existing implementations

Servers implementing the clarified behavior will provide better model self-recovery while continuing to work with all existing clients.

References

- MCP Tools Error Handling Specification
- Better MCP tools/call Error Responses: Help Your AI Recover Gracefully
- Related Issue: https://github.com/modelcontextprotocol/typescript-sdk/pull/824

SEP-1319 Decouple Request Payload from RPC Methods Definition

Final · Standards Track · Created 2025-08-08

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-08-08
- **Author(s)**: @kurtisvg
- **Issue**: #1319

Abstract

This SEP proposes a structural refactoring of the Model Context Protocol (MCP) specification. The core change is to define payload of requests (e.g., CallToolRequest) as independent definitions and have the RPC method definitions refer to these models. This decouples the definition of the data payload from the definition of the remote procedure that transports it, leading to a clearer, more modular, and more maintainable specification.

Motivation

The current MCP specification tightly couples the data payload of a request with the JSON-RPC method that transports it. This design presents several challenges:

- **Reduced Clarity:** It forces developers to mentally parse the JSON-RPC transport structure just to understand the core data being exchanged. This increases cognitive load and makes the specification difficult to read and implement correctly.
- **Hindered Maintainability:** Defining data structures inline prevents their reuse across different methods, leading to redundancy and making future updates to the protocol more complex and error-prone.
- **Tightly Coupled to JSON-RPC:** Most critically, this tight coupling to JSON-RPC is the primary blocker for defining bindings for other transport protocols. To support transports like **gRPC** (which is currently a popular ask from the community), a transport-agnostic definition of its request and response messages. The current structure makes this practically impossible.

By refactoring the specification to separate the data model (the "what") from the RPC method (the "how"), this proposal will create a clearer, more modular specification. This change will immediately improve the developer experience and, most importantly, pave the way for the future evolution of MCP across multiple transports.

Specification

The proposal introduces the following principle: All data structures used as parameters (params) or results (result) for RPC methods should be defined as standalone, named schemas. The RPC method definitions will then use references to these schemas.

Current Approach (Inline Definition):

The RPC method definition contains the full structure of its parameters and results.

```ts
export interface CallToolRequest extends Request {
  method: "tools/call";
  params: {
    name: string;
    arguments?: { [key: string]: unknown };
  };
}
```

Proposed Approach (Decoupled Definition):

First, the data models for the request and response are defined as top-level schemas.

```ts
/**
 * Parameters for a `tools/call` request.
 *
 * @category tools/call
 */
export interface CallToolRequestParams extends RequestParams {
  name: string;
  arguments?: { [key: string]: unknown };
}
```

Then, the RPC method definition becomes much simpler, merely referring to these models.

```ts
export interface CallToolRequest extends Request {
  method: "tools/call";
  params: CallToolRequestParams;
}
```

Rationale

The proposed solution—separating payload definitions from the RPC method—was chosen as the most direct and non-disruptive path to achieving the goals outlined in the motivation.

This approach establishes a clear architectural boundary between two distinct concerns:

1. **The Data Layer:** The transport-agnostic payload definition (e.g., `CallToolRequestParams`), which represents the core information being exchanged.
2. **The Transport Layer:** The protocol-specific wrapper (e.g., the JSON-RPC `CallToolRequest` object), which describes how the data is sent.

This architectural separation is superior to maintaining separate, parallel specifications for each transport (e.g., one for JSON-RPC, another for gRPC), which would introduce significant maintenance overhead and risk inconsistencies.

Crucially, this design refactors the specification document itself but intentionally **leaves the on-the-wire format unchanged**. This makes the proposal fully backward-compatible, requiring no changes from existing, compliant clients and servers. In short, this change is a strategic, foundational improvement that enables future growth without penalizing the current ecosystem.

Backward Compatibility

This proposal is a **non-breaking change** for existing implementations. It is a refactoring of the *specification document itself* and does not alter the on-the-wire JSON format of the protocol messages. A client or server that is compliant with the old specification structure will remain compliant with the new one, as the resulting JSON payloads are identical.

The primary impact is on developers who read the specification and on tools that parse the specification to generate code or documentation.

SEP-1330 Elicitation Enum Schema Improvements and Standards Compliance

Final · Standards Track · Created 2025-08-11

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-08-11
- **Author(s)**: chughtapan
- **Issue**: #1330

Abstract

This SEP proposes improvements to enum schema definitions in MCP, deprecating the non-standard `enumNames` property in favor of JSON Schema-compliant patterns, and introducing additional support for multi-select enum schemas in addition to single choice schemas. The new schemas have been validated against the JSON specification.

Schema Changes: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1148 Typescript SDK Changes: https://github.com/modelcontextprotocol/typescript-sdk/pull/1077 Python SDK Changes: https://github.com/modelcontextprotocol/python-sdk/pull/1246 **Client Implementation:** https://github.com/evalstate/fast-agent/pull/324/files **Working Demo:** https://asciinema.org/a/anBvJdqEmTjw0JkKYOooQa5Ta

Motivation

The existing schema for enums uses a non-standard approach to adding titles to enumerated values. It also limits use of enums in Elicitation (and any other schema object that should adopt `EnumSchema` in the future) to a single selection model. It is a common pattern to ask the user to select multiple entries. In the UI, this amounts to the difference between using checkboxes or radio buttons.

For these reasons, we propose the following non-breaking minor improvements to the `EnumSchema` for improving user and developer experience.

- Keep the existing `EnumSchema` as "Legacy"
  - It uses a non-standard approach for adding titles to enumerated values
  - Mark it as Legacy but still support it for now.
  - As per @dsp-ant When we have a proper deprecation strategy, we'll mark it deprecated
- Introduce the distinction between Untitled and Titled enums.
  - If the enumerated values are sufficient, no separate title need be specified for each value.
  - If the enumerated values are not optimal for display, a title may be specified for each value.
- Introduce the distinction between Single and Multi-select enums.
  - If only one value can be selected, a Single select schema can be used
  - If more than one value can be selected, a Multi-select schema can be used
- In `ElicitResponse`, add array as an `additionalProperty` type
  - Allows multiple selection of enumerated values to be returned to the server

Specification

1. Mark Current `EnumSchema` with Non-Standard `enumNames` Property as "Legacy"

The current MCP specification uses a non-standard `enumNames` property for providing display names for enum values. We propose to mark `enumNames` property as legacy, suggest using `TitledSingleSelectEnum`, a standards compliant enum type we define below.

```
// Continue to support the current EnumSchema as Legacy

/**
 * Legacy: Use TitledSingleSelectEnumSchema instead.
 * This interface will be removed in a future version.
 */
export interface LegacyEnumSchema {
  type: "string";
  title?: string;
  description?: string;
  enum: string[];
  enumNames?: string[]; // Titles for enum values (non-standard, legacy)
}
```

2. Define Single Selection Enums (with Titled and Untitled varieties)

Enums may or may not need titles. The enumerated values may be human readable and fine for display. In which case an untitled implementation using the JSON Schema keyword `enum` is simpler. Adding titles requires the `enum` array to be replaced with an array of objects using `const` and `title`.

```
// Single select enum without titles
export type UntitledSingleSelectEnumSchema = {
  type: "string";
  title?: string;
  description?: string;
  enum: string[]; // Plain enum without titles
};

// Single select enum with titles
export type TitledSingleSelectEnumSchema = {
  type: "string";
  title?: string;
  description?: string;
  oneOf: Array<{
    const: string; // Enum value
    title: string; // Display name for enum value
  }>;
};

// Combined single selection enumeration
export type SingleSelectEnumSchema =
  UntitledSingleSelectEnumSchema | TitledSingleSelectEnumSchema;
```

3. Introduce Multiple Selection Enums (with Titled and Untitled varieties)

While elicitation does not support arbitrary JSON types like arrays and objects so clients can display the selection choice easily, multiple selection enumerations can be easily implemented.

```
// Multiple select enums without titles
export type UntitledMultiSelectEnumSchema = {
  type: "array";
  title?: string;
  description?: string;
  minItems?: number; // Minimum number of items to choose
  maxItems?: number; // Maximum number of items to choose
  items: {
    type: "string";
    enum: string[]; // Plain enum without titles
  };
};

// Multiple select enums with titles
export type TitledMultiSelectEnumSchema = {
  type: "array";
  title?: string;
  description?: string;
  minItems?: number; // Minimum number of items to choose
  maxItems?: number; // Maximum number of items to choose
  items: {
    oneOf: Array<{
      const: string; // Enum value
      title: string; // Display name for enum value
    }>;
  };
};

// Combined Multiple select enumeration
export type MultiSelectEnumSchema =
  UntitledMultiSelectEnumSchema | TitledMultiSelectEnumSchema;
```

4. Combine All Varieties as `EnumSchema`

The final `EnumSchema` rolls up the legacy, multi-select, and single-select schemas as one, defined as:

```
// Combined legacy, multiple, and single select enumeration
export type EnumSchema =
  SingleSelectEnumSchema | MultiSelectEnumSchema | LegacyEnumSchema;
```

5. Extend ElicitResult

The current elicitation result schema only allows returning primitive types. We extend this to include string arrays for MultiSelectEnums:

```
export interface ElicitResult extends Result {
  action: "accept" | "decline" | "cancel";
  content?: { [key: string]: string | number | boolean | string[] }; // string[] is new
}
```

Instance Schema Examples

Single-Select Without Titles (No change)

```
{
  "type": "string",
  "title": "Color Selection",
  "description": "Choose your favorite color",
  "enum": ["Red", "Green", "Blue"],
  "default": "Green"
}
```

Legacy Single Select With Titles

```
{
  "type": "string",
  "title": "Color Selection",
  "description": "Choose your favorite color",
  "enum": ["#FF0000", "#00FF00", "#0000FF"],
  "enumNames": ["Red", "Green", "Blue"],
  "default": "#00FF00"
}
```

Single-Select with Titles

```
{
  "type": "string",
  "title": "Color Selection",
  "description": "Choose your favorite color",
  "oneOf": [
    { "const": "#FF0000", "title": "Red" },
    { "const": "#00FF00", "title": "Green" },
    { "const": "#0000FF", "title": "Blue" }
  ],
  "default": "#00FF00"
}
```

Multi-Select Without Titles

```
{
  "type": "array",
  "title": "Color Selection",
  "description": "Choose your favorite colors",
  "minItems": 1,
  "maxItems": 3,
  "items": {
    "type": "string",
    "enum": ["Red", "Green", "Blue"]
  },
  "default": ["Green"]
}
```

Multi-Select with Titles

```
{
  "type": "array",
  "title": "Color Selection",
  "description": "Choose your favorite colors",
  "minItems": 1,
  "maxItems": 3,
  "items": {
    "anyOf": [
      { "const": "#FF0000", "title": "Red" },
      { "const": "#00FF00", "title": "Green" },
      { "const": "#0000FF", "title": "Blue" }
    ]
  },
  "default": ["Green"]
}
```

Rationale

1. **Standards Compliance**: Aligns with official JSON Schema specification. Standard patterns work with existing JSON Schema validators
2. **Flexibility**: Supports both plain enums and enums with display names for single and multiple choice enums.
3. **Client Implementation:** shows that the additional overhead of implementing a group of checkboxes v/s a single checkbox is minimal: https://github.com/evalstate/fast-agent/pull/324/files

Backwards Compatibility

The `LegacyEnumSchema` type maintains backwards compatible during the migration period. Existing implementations using `enumNames` will continue to work until a protocol-wide deprecation strategy is implemented, and this schema is removed.

Reference Implementation

Schema Changes: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1148 Typescript SDK Changes: https://github.com/modelcontextprotocol/typescript-sdk/pull/1077 Python SDK Changes: https://github.com/modelcontextprotocol/python-sdk/pull/1246 **Client Implementation:** https://github.com/evalstate/fast-agent/pull/324/files **Working Demo:** https://asciinema.org/a/anBvJdqEmTjw0JkKYOooQa5Ta

Security Considerations

No security implications identified. This change is purely about schema structure and standards compliance.

Appendix

Validations

Using stored validations in the JSON Schema Validator at https://www.jsonschemavalidator.net/ we validate:

- All of the example instance schemas from this document against the proposed JSON meta-schema `EnumSchema` in the next section.
- Valid and invalid values against the example instance schemas from this document.

Legacy Single Selection

- `EnumSchema` validating a legacy single select instance schema with titles
- The legacy titled single select instance schema validating a correct single selection
- The legacy titled single select instance schema validating an incorrect single selection

Single Selection

- `EnumSchema` validating a single select instance schema without titles
- `EnumSchema` validating a single select instance schema with titles
- The untitled single select instance schema validating a correct single selection
- The untitled single select instance schema invalidating an incorrect single selection
- The titled single select instance schema validating a correct single selection
- The titled single select instance schema invalidating an incorrect single selection

Multiple Selection

- `EnumSchema` validating the multi-select instance schema without titles
- `EnumSchema` validating the multi-select instance schema with titles
- The untitled multi-select instance schema validating a correct multiple selection The untitled multi-select instance schema validating invalidating an incorrect multiple selection The titled multi-select instance schema validating a correct multiple selection The titled multi-select instance schema validating invalidating an incorrect multiple selection

JSON meta-schema

This is our proposal for the replacement of the current `EnumSchema` in the specification's `schema.json`.

```
{
  "$schema": "https://json-schema.org/draft-07/schema",
  "definitions": {
    // New Definitions Follow
    "UntitledSingleSelectEnumSchema": {
      "type": "object",
      "properties": {
        "type": { "const": "string" },
        "title": { "type": "string" },
        "description": { "type": "string" },
        "enum": {
          "type": "array",
          "items": { "type": "string" },
          "minItems": 1
        }
      },
      "required": ["type", "enum"],
      "additionalProperties": false
    },

    "UntitledMultiSelectEnumSchema": {
      "type": "object",
      "properties": {
        "type": { "const": "array" },
        "title": { "type": "string" },
        "description": { "type": "string" },
        "minItems": {
          "type": "number",
          "minimum": 0
        },
        "maxItems": {
          "type": "number",
          "minimum": 0
        },
        "items": {
          "type": "object",
          "properties": {
            "type": { "const": "string" },
            "enum": {
              "type": "array",
              "items": { "type": "string" },
              "minItems": 1
            }
          },
          "required": ["type", "enum"],
          "additionalProperties": false
        }
      },
      "required": ["type", "items"],
      "additionalProperties": false
    },

    "TitledSingleSelectEnumSchema": {
```

```
      "type": "object",
      "required": ["type", "anyOf"],
      "properties": {
        "type": { "const": "string" },
        "title": { "type": "string" },
        "description": { "type": "string" },
        "anyOf": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["const", "title"],
            "properties": {
              "const": { "type": "string" },
              "title": { "type": "string" }
            },
            "additionalProperties": false
          }
        }
      },
      "additionalProperties": false
    },

    "TitledMultiSelectEnumSchema": {
      "type": "object",
      "required": ["type", "anyOf"],
      "properties": {
        "type": { "const": "array" },
        "title": { "type": "string" },
        "description": { "type": "string" },
        "anyOf": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["const", "title"],
            "properties": {
              "const": { "type": "string" },
              "title": { "type": "string" }
            },
            "additionalProperties": false
          }
        }
      },
      "additionalProperties": false
    },

    "LegacyEnumSchema": {
      "properties": {
        "type": {
          "type": "string",
          "const": "string"
        },
        "title": { "type": "string" },
        "description": { "type": "string" },
        "enum": {
```

```
          "type": "array",
          "items": { "type": "string" }
        },
        "enumNames": {
          "type": "array",
          "items": { "type": "string" }
        }
      },
      "required": ["enum", "type"],
      "type": "object"
    },

    "EnumSchema": {
      "oneOf": [
        { "$ref": "#/definitions/UntitledSingleSelectEnumSchema" },
        { "$ref": "#/definitions/UntitledMultiSelectEnumSchema" },
        { "$ref": "#/definitions/TitledSingleSelectEnumSchema" },
        { "$ref": "#/definitions/TitledMultiSelectEnumSchema" },
        { "$ref": "#/definitions/LegacyEnumSchema" }
      ]
    }
  }
}
```

SEP-1577 Sampling With Tools

Final · Standards Track · Created 2025-09-30

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-09-30
- **Author(s)**: Olivier Chafik (@ochafik)
- **Issue**: #1577

SEP Number	#1577
Title	Sampling With Tools
Author	Olivier Chafik
Sponsor	@bhosmer-ant
Status	Draft
Created	2025-09-29
Specification	MCP 2025-06-18
Prototype	https://github.com/modelcontextprotocol/typescript-sdk/pull/991
PR	https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1796
SDKs	https://github.com/modelcontextprotocol/python-sdk/pull/1594 https://github.com/modelcontextprotocol/typescript-sdk/pull/1101

Updates:

- *Oct 1*: renamed `tool_choice` -> `toolChoice` (+ `"none"` value); removed exotic `stopReason`s `"refusal"` & `"other"`; allowed `{CreateMessageResult,SamplingMessage}.content` to be single contents or arrays of contents;
- *Oct 6*: aligned `ToolResultContent` on `CallToolResult` (support image / audio); added "Possible Follow Ups" section.
- *Oct 10*: updated reference impl example w/ simple tool registry (unify mcp tools w/ tool loop tools, see comment below) and a "choose your own adventure" game that uses sampling w/ tools + elicitation.
- *Oct 27*: aligned `ToolResultContent.content` on `CallToolResult.content` (using ContentBlock); added `ToolResultContent._meta`
- *Nov 5*:
  - kept `stopReason` as open string but w/ redundant explicit enums for visibility
  - removed requirement to throw when `includeContext` not matching advertised `ClientCapabilities.sampling.context`
  - mitigates backwards compatibility issue of `CreateMessageResult.content` being an array of contents OR a single content by saying sampling *MUST NOT* return an array in earlier spec versions (+ acknowledging SDK updates of code w/ sampling will need small code changes)

- *Nov 7*: renamed type `ToolCallContent` to `ToolUseContent` (to match its `tool_use` type & the `toolUse` `stopReason`). SEP was approved!
- *Nov 10*: removing `disable_parallel_tool_use` / keeping for a later update as the Gemini API has no way to implement this for now.
- *Nov 11*: added extra notes about Gemini API's function calling modes & roles; requiring SamplingMessage w/ tool result contents not be mixed w/ other content types

Abstract

This SEP introduces `tools` & `toolChoice` params to `sampling/createMessage` and soft-deprecates `includeContext` (fences `thisServer` & `allServers` under a capability). This allows MCP servers to run their own agentic loops using the client's tokens (still under the user supervision), and reduces the complexity of client implementations (context support becoming explicitly optional).

Motivation

- Sampling doesn't support tool calling, although it's a cornerstone of modern agentic behaviour. Without explicit support for it, MCP servers that use Sampling can either try and emulate tool calling w/ complex prompting / custom parsing of the outputs, or are limited to simpler, non-agentic requests. Adding support for tool calling could unlock many novel use cases in the MCP ecosystem.
- Context inclusion is ambiguously defined (see this doc): it makes it particularly tricky to fully implement sampling, which along with other precautions needed for sampling (unaffected by this SEP) may have contributed to low adoption of the feature in clients (feature was introduced in the MCP Nov 2024 spec).

Please note some related work:

- MCP Sampling (@jerome3o-anthropic): extremely similar proposal:
  - Add same tools semantics,
  - Deprecate `includeContext` (doc explains why its semantics are ambiguous)
  - (goes further to suggest explicit context sharing, which is out of scope from this proposal)
- Allow Prompt/Sampling Messages to contain multiple content blocks. #198
  - In this PR we've made `{CreateMessageResult,SamplingMessage}.content` to accept a single content or an array of contents. The `result.content` change is backwards incompatible but is required to support parallel tool calls. The `SamplingMessage.content` change then makes it much more natural to write a tool loop (see example in reference implementation: toolLoopSampling.ts)

In the "Possible Follow ups" Section below, we give examples of features that were kept out of scope from this SEP but which we took care to make this SEP reasonably compatible with.

Specification

Overview

- Add traditional tool call support in CreateMessageRequest w/ `tools` (w/ JSON schemas) & `toolChoice` params, requiring a server-side tool loop
  - Sampling may now yield ToolCallBlock responses
  - Server needs to call tools by itself
  - Server calls sampling again with ToolResultParamBlock to inject tool results
  - `toolChoice.mode` can be `“auto" | "required" | "none"` to allow common structured outputs use case (see below for possible follow up improvements)
  - Fenced by new capability (`sampling { tools {} }`)
- Fix/update underspecified strings in CreateMessageResult:

    - `stopReason: “endTurn" | "stopSequence" | “toolUse" | “maxToken" | string` (explicit enums + open string for compat)
    - `role: “assistant”`
- Soft-deprecate CreateMessageRequest.params.includeContext != ‘none’ (now fenced by capability)
    - Incentivize context-free sampling implementation

Protocol changes

- `sampling/createMessage`
    - ~~MUST throw an error when `includeContext is “thisServer” | “allServers”` but `clientCapabilities.sampling.context` is missing~~
    - MUST throw an error when `tool` or `toolChoice` are defined but `clientCapabilities.sampling.tools` is missing
    - Servers SHOULD avoid `[includeContext](https://openmodelcontextprotocol.org/specification/2025-06-18/schema#createmessagerequest)` != ‘none’ `as values` “thisServer” `and` “allServers”` may be removed in future spec releases.
    - `CreateMessageRequest.messages` MUST balance any “assistant” message w/ a `ToolUseContent` (and `id: $id1`) w/ a “user” message w/ a ToolResultContent (and `tool_result_id: $id1`)
        - Note: this is a requirement for Claude API implementation (parallel tool call must all be responded to in one go)
    - SamplingMessage with tool result content blocks MUST NOT contain other content types.

Schema changes

- ClientCapabilities

```
interface ClientCapabilities {
  ...
  sampling?: {
    context?: object; // NEW: Allows CreateMessageRequest.params.includeContext != "none"
    tools?: object;   // NEW: Allows CreateMessageRequest.params.{tools,toolChoice}
  };
}
```

- CreateMessageRequest (use existing Tool)

```
interface CreateMessageRequest {
  method: “sampling/createMessage”;
  params: {
    ...
    messages: SamplingMessage[]; // Note: type updated, see below

    tools?: Tool[] // NEW (existing type)

    toolChoice?: ToolChoice // NEW
  };
}

interface ToolChoice { // NEW
  mode?: “auto” | "required" | "none";
  // disable_parallel_tool_use?: boolean; // Update (Nov 10): removed, see below
}
```

    - Notes:
        - OpenAI vs. Anthropic API idioms to avoid parallel tool calls:
            - OpenAI: `parallel_tool_calls: false` (top-level param)
            - Anthropic: `tool_choice.disable_parallel_tool_use: true`
                - Preferred here as default value if unset is false (e.g. parallel tool calls allowed)
        - OpenAI vs. Anthropic API re/ `tool_choice` `"none"` vs. `tools`:
            - OpenAI: `tools: [$Foo], tool_choice: "none"` forbids any tool call
                - Preferred behaviour here
            - Anthropic: `tools: [$Foo], tool_choice: {mode: "none"}` may still call tool `Foo`
        - Gemini vs. OAI / Anthropic re/ `disable_parallel_tool_use`:
            - Gemini API has no way to disable parallel tool calls atm (unlike OAI / Anthropic APIs). Removing this flag for now, to be reintroduced when Gemini has any way of supporting it. Otherwise clients would get unexpected multiple tool calls (or alternatively if implemented that way, unexpected failures / costly retry until a single tool call is emitted)
            - Gemini API's Function calling modes have an `ANY` value that should match the proposed `required`

- SamplingMessage:

```
/*
  BEFORE:

  interface SamplingMessage {
    content: TextContent | ImageContent | AudioContent
    role: Role;
  }
*/

type SamplingMessage = UserMessage | AssistantMessage; // NEW

type AssistantMessageContent =
  TextContent | ImageContent | AudioContent | ToolUseContent;
type UserMessageContent =
  TextContent | ImageContent | AudioContent | ToolResultContent;
interface AssistantMessage {
  // NEW
  role: "assistant";
  content: AssistantMessageContent | AssistantMessageContent[];
}

interface ToolUseContent {
  // NEW
  type: "tool_use";
  name: string;
  id: string;
  input: object;
}

interface UserMessage {
  // NEW
  role: "user";
  content: UserMessageContent | UserMessageContent[];
}

interface ToolResultContent {
  // NEW
  _meta?: { [key: string]: unknown };
  type: "tool_result";
  toolUseId: string;
  content: ContentBlock[];
  structuredContent: object;
  isError?: boolean;
}
```

- Notes:
  - Differences of role vs. content type when it comes to tool calling between APIs:
    - OpenAI: `role: “system" | “user" | “assistant" | “tool"` (where tool is for tool results), while tool calls are nested in assistant messages, content is then typically null but some “OpenAI compatible” APIs accept non-null values

```
[
  { role: "user", content: "what is the temperature in london?" },
  {
    role: "assistant",
    content: "Let me use a tool...",
    tool_calls: [
      {
        id: "call_1",
        type: "function",
        function: {
          name: "get_weather",
          arguments: '{"location": "London"}',
        },
      },
    ],
  },
  {
    role: "tool",
    content: '{"temperature": 20, "condition": "sunny"}',
    tool_call_id: "call_1",
  },
];
```

- Claude API: `role: “user" | “assistant"`, tool use and result are passed through specially-typed message content parts:

```
[
  {
    "role": "user",
    "content": [
      {
        "type": "text",
        "text": "what is the temperature in london?"
      }
    },
  {
    "role": "assistant",
    "content": [
      {
        "type": "text",
        "text": "Let me use a tool..."
      },
      {
        "type": "tool_use",
        "id": "call_1",
        "name": "get_weather",
        "input": {"location": "London"}
      }
    ]
  },
  {
    "role": "user",
    "content": [
      {
        "type": "tool_result",
        "tool_call_id": "call_1",
        "content": {"temperature": 20, "condition": "sunny"}
      }
    ]
  }
]
```

- Gemini API:
  - `function` role (similar to OAI's `tool` role)
  - No tool call id concept (function calling: Gemini requires tool results to be provided in the exact same order as the tool use parts. An implementation could generate the tool call ids and use them to reorder the tool results if needed.

- CreateMessageResult

```
/*
  BEFORE:

  interface CreateMessageResult {
    _meta?: { [key: string]: unknown };
    content: TextContent | ImageContent | AudioContent;
    role: Role;
    stopReason?: string;
    [key: string]: unknown;
}
*/
interface CreateMessageResult {
  _meta?: { [key: string]: unknown };

  content: AssistantMessageContent | AssistantMessageContent[] // UPDATED

  role: "assistant"; // UPDATED

  stopReason?: “endTurn" | "stopSequence" | “toolUse" | “maxToken" | string //
UPDATED

  [key: string]: unknown;
}
```

- Notes:
  - Backwards compatibility issue: returning CreateMessageResult.content as an array of contents OR a single content is problematic, so we propose:
    - `sampling/createMessage` MUST NOT return an array in `CreateMessageResult.content` before spec version Nov 2025.
      - This guarantees wire-level backwards-compatibility
    - Existing code that uses sampling may break w/ new SDK releases as it will need to test content to know if it's an array or a single block, and act accordingly.
    - This seems reasonable(?)
  - `CreateMessageResult.stopReason` field is currently defined as an open `string`, and the spec only mentions the `endTurn` as example value.
  - OpenAI vs. Anthropic API idioms
    - Finish/stop reason
      - OpenAI’s ChatCompletion: `finish_reason: “stop” | “length” | “tool_use”` (...?)
      - Anthropic: `stop_reason: “end_turn” | “max_tokens” | “stop_sequence” | “tool_use” | “pause_turn” | “refusal”`

Possible Follow ups

These are out of scope for this SEP, but care was taken not to preclude them, so where appropriate we give examples of how they could be implemented on top of / after this SEP.

Streaming support

See: Streaming tool use results #117

This could be important for some longer-running use cases or when latency is important, but would play better w/ streaming support in MCP tools.

A possible way to implement this would be to use notifications w/ payload, and possibly create a new method `sampling/createMessageStreamed`. Both should be orthogonal w/ this SEP (but we'd need to create delta types for results, similar to streaming APIs in inference API such as Claude API and OpenAI API).

Cache friendliness updates

Two bits needed here:

- Introduce cache awareness
  - Implicit caching guidelines phrased as SHOULDs
  - Explicit cache points and TTL semantics as in the Claude API? (incl. beta behaviour for longer caching)
    - Pros: easy to implement *for at least 1 implementor (Anthropic)*
    - Cons: if hard to implement for others, unlikely to get approval.
  - “Whole prompt” / prompt-prefix cache w/ an explicit key as in the OpenAI API?
    - Pros:
      - simpler for users (no need to think about where the shared prefix stops)
      - implicitly supports updating the cache (maybe even as subtree)
    - Cons: possibly harder to implement / more storage inefficient
- Introduce allowed_tools feature to enable / disable tools w/o breaking context caching
  - Relevant to this SEP as we may want to merge this feature under the tool_choice field, similar to what OpenAI did.

```
interface ToolChoice { // NEW
  mode?: “auto” | "required";
  allowed_tools?: string[]
}
```

Allow client to call the server’s tools by itself in an agentic loop

From the server’s perspective, that would remove the need to call tools by itself / inject tool results in follow up sampling calls.

The MCP server would just allowlist its own tools in the sampling request, w/t a dedicated tool definition such as:

```
{
  type: "server-tool"; // MCP tool from same server.
  name: string;
}
```

Pros:

- Safe, limited to that server’s tools.
- If we propagate the mcp-session-id, can leverage keep any server-side session context / caching

Allow client to call any other MCP servers’ tools by itself in an agentic loop

Although this sounds similar to the previous one (allow only same server’s tools), this option wouldn’t need a protocol change / could be entirely done by the client as an implementation detail of their sampling support.

The end user would allowlist tools from any other MCP server for use in a sampling request, without the server having to ask for anything. The client UI would e.g. display a tool selection UI as part of the sampling approval

flow, auto enabling tools from same server by default.

Pros:

- Technically no spec change needed (if anything, mention this as a freedom clients have)
- Possibly similar to what CreateMessageRequest.params.includeContext = thisServer / allServers intended semantics may have meant
  - `CreateMessageRequest.params.allowImplicitToolCalls = “none” | “thisServer” | “allServers”` (assuming we wanted to give the server any control over this)

Cons:

- Classifier might be needed to avoid High potential for privacy leaks / abuse
  - If user approves Gmail MCP tool usage / delegation by mistake, server gets access to their private emails through sampling

Allow server to list & call clients’ tools (client/server → p2p)

If we say the client can now expose tools that the server can call, it opens a set of possibilities:

- The client can “forward” other servers’ tools (maybe w/ some namespacing for seamless aggregation)
  - The server can then call these tools as part of its tool loop.
- Client & Server semantics start to lose weight, we enter a more peer-to-peer, symmetrical relationship
  - Client could also ask a server for sampling, while we’re at it
  - Symmetry at the protocol layer, but still directionality at the transport layer (e.g. for HTTP transport, direction of POST requests still matters)

Simplify structured outputs use case

A major use case of sampling is to get outputs that conform to a given schema.

This is possible in OpenAI’s API for instance.

The most common workaround is to give a single tool and set `tool_choice: "required"`, which guarantees the output is a ToolCall containing inputs that conform to the tool’s input schema.

While this SEP proposes we enable this `"required"`-based workaround, as a follow up it would be great to provide more explicit / simpler JSON schema support, which would also allow schema types not allowed in tool inputs (which require an object w/ properties, so one has to pick at least a name for their outputs, which requires thinking / interplay w/ the prompting strategy):

```
interface CreateMessageRequest {
  method: “sampling/createMessage”;
  params: {
    messages: SamplingMessage[];
    ...
    format: {
      type: "json_schema",
      "schema": {
        "type": "array",
        "minItems": 5,
        "maxItems": 100
      }
    }
  }
```

SEP-1613 Establish JSON Schema 2020-12 as Default Dialect for MCP

Final · Standards Track · Created 2025-10-06

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-10-06
- **Author(s)**: Ola Hungerford
- **Issue**: #1613

Abstract

This SEP establishes JSON Schema 2020-12 as the default dialect for embedded schemas within MCP messages (tool `inputSchema` / `outputSchema` and elicitation `requestedSchema` fields). Schemas may explicitly declare alternative dialects via the `$schema` field. This resolves ambiguity that has caused compatibility issues between implementations.

Motivation

The MCP specification does not explicitly state which JSON Schema version to use for embedded schemas. This has caused:

- Validation failures between clients and servers assuming different versions
- Implementation divergence across SDK ecosystems
- Developer uncertainty requiring arbitrary version choices

Community discussion (GitHub Discussion #366, PR #655) revealed that implementations were split between draft-07 and 2020-12, with multiple maintainers and community members expressing strong preference for 2020-12 as the default.

Specification

1. Default Dialect

Embedded JSON schemas within MCP messages **MUST** conform to JSON Schema 2020-12 when no `$schema` field is present.

2. Explicit Dialect Declaration

Schemas **MAY** include an explicit `$schema` field to declare a different dialect:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "type": "object",
  "properties": {
    "name": { "type": "string" }
  }
}
```

3. Schema Validation Requirements

- Schemas **MUST** be valid according to their declared or default dialect
- The `inputSchema` field **MUST NOT** be `null`

For tools with no parameters, use one of these valid approaches:

- `true` - accepts any input (most permissive)
- `{}` - equivalent to `true`, accepts any input
- `{ "type": "object" }` - accepts any object with any properties
- `{ "type": "object", "additionalProperties": false }` - accepts only empty objects `{}`

Example for a tool with no parameters:

```
{
  "name": "get_current_time",
  "description": "Returns the current server time",
  "inputSchema": {
    "type": "object",
    "additionalProperties": false
  }
}
```

4. Scope of Application

This specification applies to:

- `tools/list` response: `inputSchema` and `outputSchema`
- `prompts/elicit` request: `requestedSchema`
- Future MCP features embedding JSON Schema definitions

5. Implementation Requirements

Servers MUST:

- Generate schemas conforming to 2020-12 by default
- Include explicit `$schema` when using non-default dialects

Clients MUST:

- Validate schemas according to declared or default dialect
- Support at least JSON Schema 2020-12

Rationale

Why 2020-12?

1. **Ecosystem alignment**: Python SDK (via Pydantic) and Go SDK implementations prefer/use 2020-12
2. **Modern features**: Better validation capabilities and composition support
3. **Community preference**: Multiple maintainers and community members in PR #655 discussion advocated for 2020-12 over draft-07
4. **Current standard**: 2020-12 is the stable version as of 2025

Why allow explicit declaration?

- Supports migration paths for existing schemas
- Provides flexibility without protocol changes
- Follows JSON Schema best practices

Alternatives considered

- **Draft-07 as default**: Rejected after community feedback; older version with less capability
- **No default**: Rejected as unnecessarily verbose; adds boilerplate
- **Multiple equal versions**: Rejected; creates unpredictability and fragmentation

Backward Compatibility

This is technically a **clarification**, and not a breaking change:

- Existing schemas without `$schema` default to 2020-12
- Servers can add explicit `$schema` during transition
- Basic schemas (type, properties, required) work across versions

Migration may be needed for schemas assuming draft-07 by default:

- Schemas using `dependencies` (→ `dependentSchemas` + `dependentRequired`)
- Positional array validation (→ `prefixItems`)

Migration strategy: Add explicit `$schema: "http://json-schema.org/draft-07/schema#"` during transition, then update to 2020-12 features.

Reference Implementation

SDK Implementations

Python SDK - Already compatible:

- Uses Pydantic for schema generation
- Pydantic defaults to 2020-12 via `.model_json_schema()`

Go SDK - Implemented 2020-12:

- Explicit 2020-12 implementation completed
- Confirmed by @samthanawalla in PR #655 discussion

Other SDKs:

- May require updates but based on other examples, there should be straightforward or out-of-the-box options to support this. I can add more examples here or we can create issues to follow up on these after acceptance.

Security Implications

No specific security implications have been identified from establishing 2020-12 as the default dialect. The clarification reduces ambiguity that could lead to validation mismatches between implementations, which is a minor security improvement through increased predictability.

Implementations should use well-maintained JSON Schema validator libraries and keep them updated, as with any dependency.

Related Work

SEP-1330: Elicitation Enum Schema Improvements

SEP-1330 proposes deprecating the non-standard `enumNames` property in favor of JSON Schema 2020-12 compliant patterns. This work is directly enabled by establishing 2020-12 as the default dialect.

Implementation Consideration:
As noted in SEP-1330 discussion, there is some concern about parsing complexity with advanced JSON Schema features like `oneOf` and `anyOf`. However, these features are part of the JSON Schema standard and well-supported by mature validator libraries. Implementations can balance standards compliance with their parsing needs by using well-tested JSON Schema validation libraries.

SEP-834: Full JSON Schema 2020-12 Support

This SEP establishes the foundation (default dialect) while SEP-834 addresses comprehensive support for 2020-12 features.

Open Questions

The schema for the spec itself references `draft-07` and the `typescript-json-schema` package we use to generate it only supports draft-07.

Options:

1. Update schema generation script to patch to 2020-12 after generation (this is what I did in the current PR)
2. Switch to a different schema generator that supports 2020-12
3. Leave as-is since it doesn't actually conflict with the spec?

Personally I'd prefer (1) in the short term and then (2) as a follow-up.

SEP-1686 Tasks

Final · Standards Track · Created 2025-10-20

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-10-20
- **Author(s)**: Surbhi Bansal, Luca Chang
- **Issue**: #1686

Abstract

> This SEP is preserved as a historical record of the experimental tasks feature shipped in the `2025-11-25` specification. The code examples below are non-normative pseudocode written against the v1 SDKs. The draft specification moves tasks out of the core protocol and into the `io.modelcontextprotocol/tasks` extension (SEP-2663).

This SEP improves support for task-based workflows in the Model Context Protocol (MCP). It introduces both the **task primitive** and the associated **task ID**, which can be used to query the state and results of a task, up to a server-defined duration after the task has completed. This primitive is designed to augment other requests (such as tool calls) to enable call-now, fetch-later execution patterns across all requests for servers that support this primitive.

Motivation

The current MCP specification supports tool calls that execute a request and eventually receive a response, and tool calls can be passed a progress token to integrate with MCP's progress-tracking functionality, enabling host applications to receive status updates for a tool call via notifications. However, there is no way for a client to explicitly request the status of a tool call, resulting in states where it is possible for a tool call to have been dropped on the server, and it is unknown if a response or a notification may ever arrive. Similarly, there is no way for a client to explicitly retrieve the result of a tool call after it has completed — if the result was dropped, clients must call the tool again, which is undesirable for tools expected to take minutes or more. This is particularly relevant for MCP servers abstracting existing workflow-based APIs, such as AWS Step Functions, Workflows for Google Cloud, or APIs representing CI/CD pipelines, among other applications.

Today, it is possible for individual MCP servers to represent tools in a way that enables this, with certain compromises. For example, a server may expose a `long_running_tool` and wish to support this pattern, splitting it into three separate tools to accommodate this:

1. `start_long_running_tool`: This would start the work represented by `long_running_tool` and return a tracking token of some kind, such as a job ID.
2. `get_long_running_tool_status(token)`: This would accept the tracking token and return the current status of the tool call, informing the caller that the operation is still ongoing.
3. `get_long_running_tool_result(token)`: This would accept the tracking token and return the result of the tool call, if it is available.

Representing a tool in this way seems to solve for the use case, but it introduces a new problem: Tools are generally-expected to be orchestrated by an agent, and agent-driven polling is both unnecessarily expensive and inconsistent — it relies on prompt engineering to steer an agent to poll at all. In the original `long_running_tool` case, the client had no way of knowing if a response would ever be received, while in the `start_long_running_tool` case, the application has no way of knowing if the agent will orchestrate tools according to the specific contract of the server.

It is also impossible for the host application to take ownership of this orchestration, as this tool-splitting is both conventions-based and may be implemented in different ways across MCP servers — one server may have three tools for one conceptual operation (as in our example), or it may have more, in the case of more complex, multi-step operations.

On the other hand, if active task polling is not needed, existing MCP servers can fully-wrap a workflow API in a single tool call that polls for a result, but this introduces an undesirable implementation cost: an MCP server wrapping an existing workflow API is a server that only exists for polling other systems.

Affected Customer Use Cases These concerns are backed by real use cases that Amazon has seen both internally and with their external customers (identities redacted where non-public):

1. Healthcare & Life Sciences Data Analysis *Challenge:* Amazon's customers in the healthcare and life sciences industry are attempting to use MCP to wrap existing computational tools to analyze molecular properties and predict drug interactions, processing hundreds of thousands of data points per job from chemical libraries through multiple inference models simultaneously. These complex, multi-step workflows require a way to actively check statuses, as they take upwards of several hours, making retries undesirable. ***Current Workaround:*** Not yet determined. ***Impact:*** Cannot integrate with real-time research workflows, prevents interactive drug discovery platforms, and blocks automated research pipelines. These customers are looking for best practices for workflow-based tool calls and have noted the lack of first-class support in MCP as a concern. If these customers do not have a solution for long-running tool calls, they will likely forego MCP and continue using their existing platforms. ***Ideal:*** Concurrent and poll-able tool calls as an answer for operations executing in the range of a few minutes, and some form of push notification system to avoid blocking their agents on long analyses on the order of hours. This SEP supports the former use case, and offers a framework that could extend to support the latter.

2. Enterprise Automation Platforms *Challenge:* Amazon's large enterprise customers are looking to develop internal MCP platforms to automate SDLC processes across their organizations, extending to sales, customer service, legal, HR, and cross-divisional teams. They have noted they have long-running agent and agent-tool interactions, supporting complex business process automation. ***Current Workaround:*** Not yet determined. Considering an application-level system outside of MCP backed by webhooks. ***Impact:*** Limitations related to the host application being unaware of tool execution state prevent complex business process automation and limit sophisticated multi-step operations. These customers want to dispatch processes concurrently and collect their results later, and are noting the lack of explicit late-retrieval as a concern — and are considering involved application-level notification systems as a possible workaround. ***Ideal:*** Built-in mechanisms for actively checking the status of ongoing work to avoid needing to implement notification systems specific to their own tool conventions themselves.

3. Code Migration Workflows *Challenge*: Amazon has automated code migration and transformation tools to perform upgrades across its own codebases and those of external customers, and is attempting to wrap those tools in MCP servers. These migrations analyze dependencies, transform code to avoid deprecated runtime features, and validate changes across multiple repositories. These migrations range from minutes to hours depending on migration scope, complexity, and validation requirements. ***Current Workaround:*** Developers implement manual tracking by splitting a job into `create` and `get` tools, forcing models to manage state and repeatedly poll for completion. ***Impact:*** Poor developer experience due to needing to replicate this hand-rolled polling mechanism across many tools. One team had to debug an issue where the model would hallucinate job names if it hadn't listed them first. Validating that this does not happen across many tools in a large toolset is time-consuming and error-prone. ***Ideal:*** Support natively polling tool state at the data layer to support pushing a tool to the background and avoiding blocking other tasks in the chat session, while still supporting deterministic

polling and result retrieval. The team needs the same pattern across many tools in their MCP servers, and wants a common solution across them, which this SEP directly supports.

4. Test Execution Platforms *Challenge:* Amazon's internal test infrastructure executes comprehensive test suites including thousands of cases, integration tests across services, and performance benchmarks. They have built an MCP server wrapping this existing infrastructure. ***Current Workaround:*** For streaming test logs, the MCP server exposes a tool that can read a range of log lines, as it cannot effectively notify the client when the execution is complete. There is not yet any workaround for executing test runs. ***Impact:*** Cannot run a test suite and stream its logs simultaneously without a single hours-long tool call, which would time out on either the client or the server. This prevents agents from looking into test failures in an incomplete test run until the entire test suite has completed, potentially hours later. ***Ideal:*** Support host application-driven tool polling for intermediate results, so a client can be notified when a long-running tool is complete. This SEP does not fully-support this use case (it does enable polling), but the Task execution model can be extended to do so, as discussed in the "Future Work" section.

5. Deep Research *Challenge:* Deep research tools spawn multiple research agents to gather and summarize information about topics, going through several rounds of search and conversation turns internally to produce a final result for the caller application. The tool takes an extended amount of time to execute, and it is not always clear if the tool is still executing. ***Current Workaround:*** The research tool is split into a separate `create` tool to create a report job and a `get` tool to get the status/result of that job later. ***Impact:*** When using this with host applications, the agent sometimes runs into issues calling the `get` tool repeatedly — in particular, it calls the tool once before ending its conversation turn, claiming to be "waiting" before calling the tool again. It cannot resume until receiving a new user message. This also complicates expiration times, as it is not possible to predict when the client will retrieve the result when this occurs. It is possible to work around this by adding a `wait` tool for the model, but this prevents the model from doing anything else concurrently. ***Ideal:*** Support polling a tool call's state in a deterministic way and notify the model when a result is ready, so the tool result can be immediately retrieved and deleted from the server. Other than notifying the model (a host application concern), this SEP fully supports this use case.

6. Agent-to-Agent Communication (Multi-Agent Systems) *Challenge:* One of Amazon's internal multi-agent systems for customer question answering faces scenarios where agents require significant processing time for complex reasoning, research, or analysis. When agents communicate through MCP, slow agents cause cascading delays throughout this system, as agents are forced to wait on their peers to complete their work. ***Current Workaround:*** Not yet determined. ***Impact:*** Communication pattern creates cascading delays, prevents parallel agent processing, and degrades system responsiveness for other time-sensitive interactions. ***Ideal:*** Some method to allow agents to perform other work concurrently and get notified once long-running tasks complete. This SEP supports this use case by enabling host applications to implement background polling for select tool calls without blocking agents.

These use cases demonstrate that a mechanism to actively track tool calls and defer results is a real requirement for these types of MCP deployments in production environments.

Integration with Existing Architectures Many workflow-driven systems already provide active execution-tracking capabilities with built-in status metadata, monitoring, and data retention policies. This proposal enables MCP servers to expose these existing APIs with thin MCP wrappers while maintaining their existing reliability.

Benefits for Existing Architectures:

- **Leverage Existing State Management:** Systems like AWS Step Functions, Workflows for Google Cloud, and CI/CD platforms already maintain execution state, logs, and results. MCP servers can expose these systems' existing APIs without pushing the responsibility of polling to a fallible agent.
- **Preserve Native Monitoring:** Existing monitoring, alerting, and observability tools continue to work unchanged. The execution happens almost entirely within the existing workflow-management system.
- **Reduce Implementation Overhead:** Server implementers don't need to build new state management, persistence, or monitoring infrastructure. They can focus on the MCP protocol mapping of their existing APIs to tasks.

This SEP simplifies integration with existing workflows and allows workflow services to continue to manage their own state while delivering a quality customer experience, rather than offloading to agent-polling or building MCP servers that do nothing but poll other services.

Specification

This SEP introduces a mechanism for requestors (which can be either clients or servers, depending on the direction of communication) to augment their requests with **tasks**. Tasks are durable state machines that carry information about the underlying execution state of the request they wrap, and are intended for requestor polling and deferred result retrieval. Each task is uniquely identifiable by a requestor-generated **task ID**.

1. User Interaction Model

Tasks are designed to be **application-driven**—receivers tightly-control which requests (if any) support task-based execution and manage the lifecycles of those tasks; meanwhile, requestors own the responsibility for augmenting requests with tasks, and for polling on the results of those tasks.

Implementations are free to expose tasks through any interface pattern that suits their needs—the protocol itself does not mandate any specific user interaction model.

2. Capabilities

Servers and clients that support task-augmented requests **MUST** declare a `tasks` capability during initialization. The `tasks` capability is structured by request category, with boolean properties indicating which specific request types support task augmentation.

Refer to https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1732 for details.

3. Protocol Messages

3.1. Creating Tasks

To create a task, requestors send a request with the `openmodelcontextprotocol.org/task` key included in `_meta`, with a `taskId` value representing the task ID. Requestors **MAY** include a `keepAlive`, with a value representing how long after completion the requestor would like the task results to be kept for.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "some_method",
  "params": {
    "_meta": {
      "openmodelcontextprotocol.org/task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
        "keepAlive": 60000
      }
    }
  }
}
```

3.2. Getting Tasks

To retrieve the state of a task, requestors send a `tasks/get` request:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "keepAlive": 30000,
    "pollFrequency": 5000,
    "status": "submitted",
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

3.3. Retrieving Task Results

To retrieve the result of a completed task, requestors send a `tasks/result` request:

Request:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tasks/result",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly
cloudy"
      }
    ],
    "isError": false,
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

3.4. Task Creation Notification

When a receiver creates a task, it **MUST** send a `notifications/tasks/created` notification to inform the requestor that the task has been created and polling can begin.

Notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tasks/created",
  "params": {
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

The task ID is conveyed through the `openmodelcontextprotocol.org/related-task` metadata key. The notification parameters are otherwise empty.

This notification resolves the race condition where a requestor might attempt to poll for a task before the receiver has finished creating it. By sending this notification immediately after task creation, the receiver signals that the task is ready to be queried via `tasks/get`.

Receivers that do not support tasks (and thus ignore task metadata in requests) will not send this notification, allowing requestors to fall back to waiting for the original request response.

3.5. Listing Tasks

To retrieve a list of tasks, requestors send a `tasks/list` request. This operation supports pagination.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "tasks/list",
  "params": {
    "cursor": "optional-cursor-value"
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "tasks": [
      {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
        "status": "working",
        "keepAlive": 30000,
        "pollFrequency": 5000
      },
      {
        "taskId": "abc123-def456-ghi789",
        "status": "completed",
        "keepAlive": 60000
      }
    ],
    "nextCursor": "next-page-cursor"
  }
}
```

3.6 Deleting Tasks

To explicitly delete a task and its associated results, requestors send a `tasks/delete` request.

Request:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "method": "tasks/delete",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

Response:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

4. Behavior Requirements

These requirements apply to all parties that support receiving task-augmented requests.

4.1. Task Support and Handling

1. Receivers that do not support task augmentation on a request **MUST** process the request normally, ignoring any task metadata in `_meta`.
2. Receivers that support task augmentation **MAY** choose which request types support tasks.

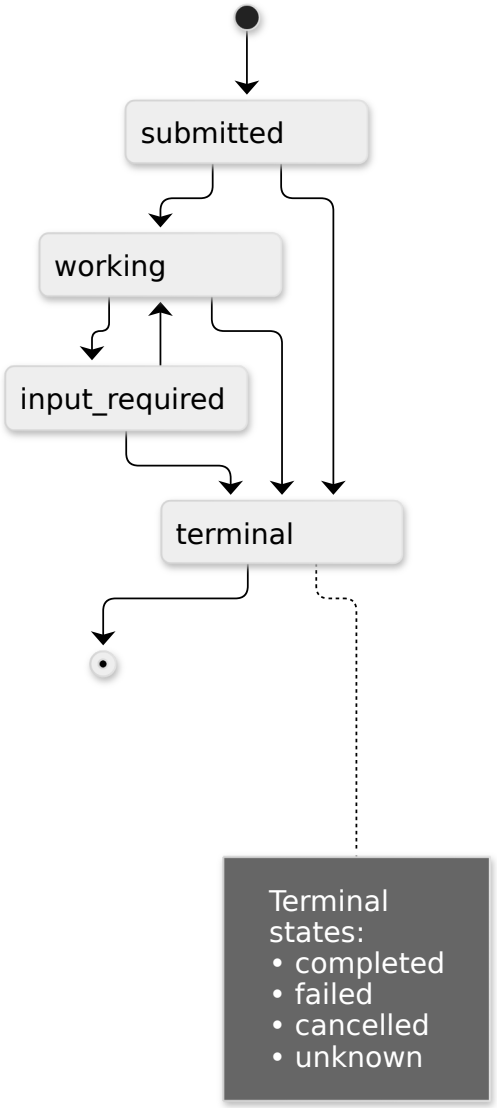
4.2. Task ID Requirements

1. Task IDs **MUST** be a string value.
2. Task IDs **SHOULD** be unique across all tasks controlled by the receiver.
3. The receiver of a request with a task ID in its `_meta` **MAY** validate that the provided task ID has not already been associated with a task controlled by that receiver.

4.3. Task Status Lifecycle

1. Tasks **MUST** begin in the `submitted` status when created.
2. Receivers **MUST** only transition tasks through the following valid paths:
    1. From `submitted`: may move to `working`, `input_required`, `completed`, `failed`, `cancelled`, or `unknown`
    2. From `working`: may move to `input_required`, `completed`, `failed`, `cancelled`, or `unknown`
    3. From `input_required`: may move to `working`, `completed`, `failed`, `cancelled`, or `unknown`
    4. Tasks in `completed`, `failed`, `cancelled`, or `unknown` status **MUST NOT** transition to any other status (terminal states)
3. Receivers **MAY** move directly from `submitted` to `completed` if execution completes immediately.
4. The `unknown` status is a terminal fallback state for unexpected error conditions. Receivers **SHOULD** use `failed` with an error message instead when possible.

Task Status State Diagram:



4.4. Input Required Status

1. When a receiver sends a request associated with a task (e.g., elicitation, sampling), the receiver **MUST** move the task to the `input_required` status.
2. The receiver **MUST** include the `openmodelcontextprotocol.org/related-task` metadata in the request to associate it with the task.
3. When the receiver receives all required responses, the task **MAY** transition out of `input_required` status (typically back to `working`).
4. If multiple related requests are pending, the task **SHOULD** remain in `input_required` status until all are resolved.

4.5. Keep-Alive and Resource Management

1. Receivers **MAY** override the requested `keepAlive` duration.
2. Receivers **MUST** include the actual `keepAlive` duration (or `null` for unlimited) in `tasks/get` responses.
3. After a task reaches a terminal status (`completed`, `failed`, or `cancelled`) and its `keepAlive` duration has elapsed, receivers **MAY** delete the task and its results.
4. Receivers **MAY** include a `pollFrequency` value (in milliseconds) in `tasks/get` responses to suggest polling intervals. Requestors **SHOULD** respect this value when provided.

4.6. Result Retrieval

1. Receivers **MUST** only return results from `tasks/result` when the task status is `completed`.
2. Receivers **MUST** return an error if `tasks/result` is called for a task in any other status.
3. Requestors **MAY** call `tasks/result` multiple times for the same task while it remains available.

4.7. Associating Task-Related Messages

1. All requests, notifications, and responses related to a task **MUST** include the `openmodelcontextprotocol.org/related-task` key in their `_meta`, with the value set to an object with a `taskId` matching the associated task ID.
2. For example, an elicitation that a task-augmented tool call depends on **MUST** share the same related task ID with that tool call's task.

4.8. Task Cancellation

1. When a receiver receives a `notifications/cancelled` notification for the JSON-RPC request ID of a task-augmented request, the receiver **SHOULD** immediately move the task to the `cancelled` status and cease all processing associated with that task.
2. Due to the asynchronous nature of notifications, receivers **MAY** not cancel task processing instantaneously. Receivers **SHOULD** make a best-effort attempt to halt execution as quickly as possible.
3. If a `notifications/cancelled` notification arrives after a task has already reached a terminal status (`completed`, `failed`, `cancelled`, or `unknown`), receivers **SHOULD** ignore the notification.
4. After a task reaches `cancelled` status and its `keepAlive` duration has elapsed, receivers **MAY** delete the task and its metadata.
5. Requestors **MAY** send `notifications/cancelled` at any time during task execution, including when the task is in `input_required` status. If a task is cancelled while in `input_required` status, receivers **SHOULD** also disregard any pending responses to associated requests.
6. Because notifications do not provide confirmation of receipt, requestors **SHOULD** continue to poll with `tasks/get` after sending a cancellation notification to confirm the task has transitioned to `cancelled` status. If the task does not transition to `cancelled` within a reasonable timeframe, requestors **MAY** assume the cancellation was not processed.

4.9. Task Listing

1. Receivers **SHOULD** use cursor-based pagination to limit the number of tasks returned in a single response.
2. Receivers **MUST** include a `nextCursor` in the response if more tasks are available.
3. Requestors **MUST** treat cursors as opaque tokens and not attempt to parse or modify them.
4. If a task is retrievable via `tasks/get` for a requestor, it **MUST** be retrievable via `tasks/list` for that requestor.

4.10 Task Deletion

1. Receivers **MAY** accept or reject delete requests for any task at their discretion.
2. If a receiver accepts a delete request, it **SHOULD** delete the task and all associated results and metadata.
3. Receivers **MAY** choose not to support deletion at all, or only support deletion for tasks in certain statuses (e.g., only terminal statuses).
4. Requestors **SHOULD** delete tasks containing sensitive data promptly rather than relying solely on `keepAlive` expiration for cleanup.

5. Message Flow

https://github.com/modelcontextprotocol/modelcontextprotocol/issues/1686#issuecomment-3452378176

6. Data Types

Task

A task represents the execution state of a request. The task metadata includes:

- `taskId`: Unique identifier for the task
- `keepAlive`: Time in milliseconds that results will be kept available after completion
- `pollFrequency`: Suggested time in milliseconds between status checks
- `status`: Current state of the task execution

Task Status

Tasks can be in one of the following states:

- `submitted`: The request has been received and queued for execution
- `working`: The request is currently being processed
- `input_required`: The request is waiting on additional input from the requestor
- `completed`: The request completed successfully and results are available
- `failed`: The task lifecycle itself encountered an error, unrelated to the associated request logic
- `cancelled`: The request was cancelled before completion
- `unknown`: A terminal fallback state for unexpected error conditions when the receiver cannot determine the actual task state

Task Metadata

When augmenting a request with task execution, the `openmodelcontextprotocol.org/task` key is included in `_meta`:

```
{
  "openmodelcontextprotocol.org/task": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "keepAlive": 60000
  }
}
```

Fields:

- `taskId` (string, required): Client-generated unique identifier for the task
- `keepAlive` (number, optional): Requested duration in milliseconds to retain results after completion

Task Creation Notification

When a receiver creates a task, it sends a `notifications/tasks/created` notification to signal that the task is ready for polling. The notification has empty params, with the task ID conveyed through the `openmodelcontextprotocol.org/related-task` metadata key:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tasks/created",
  "params": {
    "_meta": {
      "openmodelcontextprotocol.org/related-task": {
        "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
      }
    }
  }
}
```

This notification enables requestors to begin polling without encountering race conditions where the task might not yet exist on the receiver.

Task Get Request

The `tasks/get` request retrieves the current state of a task:

```
{
  taskId: string; // The task identifier to query
}
```

Task Get Response

The `tasks/get` response includes:

```
{
  taskId: string; // The task identifier
  status: TaskStatus; // Current task state
  keepAlive: number | null; // Actual retention duration in milliseconds, null for
unlimited
  pollFrequency?: number; // Suggested polling interval in milliseconds
  error?: string; // Error message if status is "failed"
}
```

Task Result Request

The `tasks/result` request retrieves the result of a completed task:

```
{
  taskId: string; // The task identifier to retrieve results for
}
```

Task Result Response

The `tasks/result` response returns the original result that would have been returned by the request:

```
{
  // The structure matches the result type of the original request
  // For example, a tools/call task would return CallToolResult structure
  [key: string]: unknown;
}
```

The result structure depends on the original request type. The receiver returns the same result structure that would have been returned if the request had been executed without task augmentation.

Task List Request

The `tasks/list` request retrieves a list of tasks:

```
{
  cursor?: string; // Optional cursor for pagination
}
```

Task List Response

The `tasks/list` response includes:

```
{
  tasks: Array<{
    taskId: string;           // The task identifier
    status: TaskStatus;       // Current task state
    keepAlive: number | null; // Retention duration in milliseconds, null for unlimited
    pollFrequency?: number;   // Suggested polling interval in milliseconds
    error?: string;           // Error message if status is "failed"
  }>;
  nextCursor?: string;        // Cursor for next page, absent if no more results
}
```

Related Task Metadata

All requests, responses, and notifications associated with a task **MUST** include the `openmodelcontextprotocol.org/related-task` key in `_meta`:

```
{
  "openmodelcontextprotocol.org/related-task": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

This associates messages with their originating task across the entire request lifecycle.

7. Error Handling

Tasks use two error reporting mechanisms:

1. **Protocol Errors**: Standard JSON-RPC errors for protocol-level issues
2. **Task Execution Errors**: Errors in the underlying request execution, reported through task status

7.1. Protocol Errors

Receivers **MUST** return standard JSON-RPC errors for the following protocol error cases:

- Invalid or nonexistent `taskId` in `tasks/get`, `tasks/list`, or `tasks/result`: `-32602` (Invalid params)
- Invalid or nonexistent cursor in `tasks/list`: `-32602` (Invalid params)
- Request with a `taskId` that was already used for a different task (if the receiver validates task ID uniqueness): `-32602` (Invalid params)
- Attempting to retrieve result when task is not in `completed` status: `-32602` (Invalid params)
- Internal errors: `-32603` (Internal error)

Receivers **SHOULD** provide informative error messages to describe the cause of errors.

Example: Task not found

```
{
  "jsonrpc": "2.0",
  "id": 70,
  "error": {
    "code": -32602,
    "message": "Failed to retrieve task: Task not found"
  }
}
```

Example: Task expired

```
{
  "jsonrpc": "2.0",
  "id": 71,
  "error": {
    "code": -32602,
    "message": "Failed to retrieve task: Task has expired"
  }
}
```

> NOTE: Receivers are not obligated to retain task metadata indefinitely. It is compliant behavior for a receiver to return a "not-found" error if it has purged an expired task.

Example: Result requested for incomplete task

```
{
  "jsonrpc": "2.0",
  "id": 72,
  "error": {
    "code": -32602,
    "message": "Cannot retrieve result: Task status is 'working', not 'completed'"
  }
}
```

Example: Duplicate task ID (if receiver validates uniqueness)

```
{
  "jsonrpc": "2.0",
  "id": 73,
  "error": {
    "code": -32602,
    "message": "Task ID already exists: 786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

7.2. Task Execution Errors

When the underlying request fails during execution, the task moves to the `failed` status. The `tasks/get` response **SHOULD** include an `error` field with details about the failure:

```
{
  taskId: string;
  status: "failed";
  keepAlive: number | null;
  pollFrequency?: number;
  error?: string;  // Description of what went wrong
}
```

Example: Task with execution error

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "failed",
    "keepAlive": 30000,
    "error": "Tool execution failed: API rate limit exceeded"
  }
}
```

For tasks that wrap requests with their own error semantics (like `tools/call` with `isError: true`), the task should still reach `completed` status, and the error information is conveyed through the result structure of the original request type.

8. Security Considerations

8.1. Task Isolation and Access Control

1. Receivers **SHOULD** scope task IDs to prevent unauthorized access:
    1. Bind tasks to the session that created them (if sessions are supported)
    2. Bind tasks to the authentication context (if authentication is used)
    3. Reject `tasks/get`, `tasks/list`, or `tasks/result` requests for tasks from different sessions or auth contexts

2. Receivers that do not implement session or authentication binding **SHOULD** document this limitation clearly, as task results may be accessible to any requestor that can guess the task ID.
3. Receivers **SHOULD** implement rate limiting on:
    1. Task creation to prevent resource exhaustion
    2. Task status polling to prevent denial of service
    3. Task result retrieval attempts
    4. Task listing requests to prevent denial of service

8.2. Resource Management

> WARNING: Task results may persist longer than the original request execution time. For sensitive operations, requestors should carefully consider the security implications of extended result retention and may want to retrieve results promptly and request shorter `keepAlive` durations.

1. Receivers **SHOULD**:
    1. Enforce limits on concurrent tasks per requestor
    2. Enforce maximum `keepAlive` durations to prevent indefinite resource retention
    3. Clean up expired tasks promptly to free resources
2. Receivers **SHOULD**:
    1. Document maximum supported `keepAlive` duration
    2. Document maximum concurrent tasks per requestor
    3. Implement monitoring and alerting for resource usage

8.3. Audit and Logging

1. Receivers **SHOULD**:
    1. Log task creation, completion, and retrieval events for audit purposes
    2. Include session/auth context in logs when available
    3. Monitor for suspicious patterns (e.g., many failed task lookups, excessive polling)
2. Requestors **SHOULD**:
    1. Log task lifecycle events for debugging and audit purposes
    2. Track task IDs and their associated operations

Rationale

Design Decision: Generic Task Primitive

The decision to implement tasks as a generic request augmentation mechanism (rather than tool-specific or method-specific) was made to maximize protocol simplicity and flexibility.

Tasks are designed to work with any request type in the MCP protocol, not just tool calls. This means that `resources/read`, `prompts/get`, `sampling/createMessage`, and any future request types can all be augmented with task metadata. This approach provides significant benefits over a tool-specific design.

From a protocol perspective, this design eliminates the need for separate task implementations per request type. Instead of defining different async patterns for tools versus resources versus prompts, a single set of task management methods (`tasks/get` and `tasks/result`) works uniformly across all request types. This uniformity reduces cognitive load for implementers and creates a consistent experience for applications using the protocol.

The generic design also provides implementation flexibility. Servers can choose which requests support task augmentation without requiring protocol changes or version negotiation. If a server doesn't support tasks for a particular request type, it simply ignores the task metadata and processes the request normally. This allows

servers to add task support to requests incrementally, starting with high-value operations and expanding over time based on actual usage patterns.

Architecturally, tasks are treated as metadata rather than a separate execution model. They augment existing requests rather than replacing them. The original request/response flow remains intact—the request still gets a response eventually. Tasks simply provide an additional polling-based mechanism for result retrieval. This design ensures that related messages (such as elicitations during task execution) can be associated consistently via the `openmodelcontextprotocol.org/related-task` metadata key, regardless of the underlying request type.

Design Decision: Metadata-Based Augmentation

Using `_meta` for task information rather than dedicated request parameters was chosen to maintain a clear separation of concerns between request semantics and execution tracking.

Task information is fundamentally orthogonal to request semantics. The task ID and keepAlive duration don't affect what the request does—they only affect how the result is retrieved and retained. A `tools/call` request performs the same operation whether or not it includes task metadata. The task metadata simply provides an alternative mechanism for accessing the result.

By placing task information in `_meta`, we create a clear architectural boundary between "what to execute" (request parameters) and "how to track execution" (task metadata). This boundary makes it easier for implementers to reason about the protocol. Request parameters define the operation being performed, while metadata provides orthogonal concerns like progress tracking, task management, and other execution-related information.

This approach also provides natural backward compatibility. Servers that don't support tasks can ignore the `_meta` content without breaking request processing. The request parameters remain valid and complete, so the operation can proceed normally. This means no protocol version negotiation is required—the new functionality is purely additive and non-disruptive.

SDKs can provide ergonomic abstractions over the task primitive while maintaining the separation of concerns, for example:

```
// === MCP SDK (Pseudocode based loosely on modelcontextprotocol/typescript-sdk) ===

/**
 * NEW: A request that resolves to a result, either directly or by polling a task.
 */
class PendingRequest<TResult> {
  constructor(readonly protocol: Protocol, readonly result: Promise<TResult>, readonly
taskId?: string) {}

  /**
   * Waits for a result, calling onTaskStatus if provided and a task was created.
   */
  async result({ onTaskStatus }): Promise<TResult> => {
    if (!onTaskStatus || !this.taskId) {
      // No task listener or task ID provided, just block for the result
      return await result;
    }

    // Whichever is successful first (or a failure if all fail) is returned.
    return Promise.any([
      result, // Blocks for result
      (async () => {
        // Blocks for a notifications/tasks/created with the provided task ID
        await this.protocol.waitForTask(this.taskId);
        return await taskHandler(this.taskId);
      })(),
    ]);
  }

  /**
   * Encapsulates polling for a result, calling onTaskStatus after querying the task.
   */
  private async taskHandler({ onTaskStatus }): Promise<TResult> => {
    // Poll for completion
    let task: Task;
    do {
      task = await this.protocol.getTask(this.taskId);
      await onTaskStatus(task);
      await sleep(task.pollFrequency ?? DEFAULT_POLLING_INTERNAL);
    } while (!task.isTerminal());

    // Process result
    return await this.protocol.getTaskResult(this.taskId);
  }
}

/**
 * Simplified/partial client session implementation for illustration purposes.
 * Extends a base class it shares with the server.
 */
class Client extends Protocol {
  /**
   * Existing request method, but with most implementation refactored to beginCallTool
```

```
   */
  async callTool<TResult>(
    params: CallToolRequest['params'],
    resultSchema: Schema<TResult>,
  ) {
    // Existing request methods can be changed to reuse new methods exposed for
    // separating request/response flows.
    const request = await this.beginCallTool(params, resultSchema);
    return request.result();
  }

  /**
   * NEW: Low-level method that starts a tool call and returns a PendingRequest
   * object for more granular control.
   */
  async beginCallTool<TResult>(
    params: CallToolRequest['params'],
    resultSchema: Schema<TResult>,
  ) {
    const request = await this.beginRequest({ method: 'tools/call', params },
resultSchema, options);
    return request;
  }
}

// === HOST APPLICATION ===

// Begin a tool call with task support
const pending: PendingRequest<CallToolResult> = await client.beginCallTool(
  {
    name: "analyze_dataset",
    arguments: { dataset: "large_file.csv" },
  },
  CallToolResultSchema,
  {
    keepAlive: 3600000,
  },
);

// Client code can assume tasks are supported, and the fallback case can be handled
internally
const result = await pending.result({
  onTaskStatus: async (task) => {
    await sendLatestStateSomewhere(task);
  },
});
```

As the design does not alter the basic request semantics, the existing form would continue to work as well:

```
const result = await client.callTool(
  {
    name: "analyze_dataset",
    arguments: { dataset: "large_file.csv" },
  },
  CallToolResultSchema,
);
```

Design Decision: Client-Generated Task IDs

The choice to have clients generate task IDs rather than having servers assign them provides several critical benefits:

Idempotency and Fault Tolerance: The primary benefit is enabling idempotent task creation. When a client generates the task ID, it can safely retry a task-augmented request if it doesn't receive a response, knowing that the server will recognize the duplicate task ID and return an error. This is essential for reliable operation over unreliable networks:

- If a request times out, the client can safely retry without creating duplicate tasks
- If a connection drops before the response arrives, the client can reconnect and retry
- The server validates task ID uniqueness and returns an error for duplicates, confirming whether the task was created

With server-generated task IDs, a timeout or connection failure creates uncertainty—the client doesn't know whether the task was created, and has no safe way to retry without potentially creating duplicate tasks.

Simplicity for Clients: Client-generated task IDs simplify the client's implementation by eliminating the need to correlate the initial response with a task identifier. The client can immediately begin polling for task status using the task ID it generated, without needing to parse the response to extract a server-assigned identifier. This is particularly valuable for asynchronous programming models where the client may want to store the task ID before the response arrives.

Trade-offs for Servers: The main trade-off is that servers wrapping existing workflow systems with their own task identifiers will generally handle this by maintaining a mapping between the client-provided task IDs and the underlying system's identifiers. For example, an MCP server wrapping AWS Step Functions might receive a client-generated task ID like `"client-abc-123"` and need to track that it corresponds to Step Functions execution ARN `"arn:aws:states:...:exec-xyz"`.

This requires:

- Persistent storage for the task ID mapping (typically a simple key-value store)
- Maintaining the mapping for the task's keepAlive duration
- Handling mapping lookups for task status and result retrieval

However, this complexity is typically minor compared to the overall work of integrating an existing workflow system into MCP. Most workflow systems already require state management for tracking execution, and maintaining a task ID mapping is a straightforward addition. The mapping structure is simple (client task ID maps to an internal identifier), and can be implemented using existing databases or key-value stores such a server likely already uses for other state management.

Design Decision: Task Creation Notification

The decision to use a `notifications/tasks/created` notification rather than altering the response semantics (as #1391 proposed) acknowledges the asynchronous nature of task creation and enables efficient race patterns between task-based polling and traditional request/response flows.

When a server creates a task, it must signal to the client that the task is ready for polling. There are at least two possible approaches: (1) the initial request could return synchronously with task metadata, or (2) the server could send a notification. This proposal uses notifications for several key reasons:

1. Notifications enable fire-and-forget request processing. The server can accept the request, begin processing it, and send the notification once the task is created, without needing to block the initial request/response cycle. This is particularly important for servers that dispatch work to background systems or queues—they can acknowledge the request immediately and send the notification once the background system confirms task creation.
2. Notifications support the race pattern that enables graceful degradation. Clients can race between waiting for the original request's response and waiting for the `notifications/tasks/created` notification. If the server doesn't support tasks, no notification arrives and the original response wins. If the server does support tasks, the notification typically arrives first (or approximately simultaneously), enabling polling to begin. A synchronous response would force clients to wait for the response before knowing whether to poll or not.
3. Notifications avoid ambiguity with existing protocol semantics. If the initial request response included task metadata and the client then polled for results, it would change the implied meaning of existing notification types:
    1. **Progress notifications**: The current MCP specification requires that progress notifications reference tokens that "are associated with an in-progress operation." While "operation" is not formally defined, the implied understanding is that an operation is bounded by a request/response pair—progress notifications stop when the response is sent. With a synchronous response containing task metadata, progress notifications would need to continue while the task executes, expanding the implied meaning of "operation" to include asynchronous tasks that outlive the original request/response cycle. The notification-based approach avoids this semantic expansion by keeping progress notifications tied to the initial request's lifecycle, while future task-based progress can be cleanly associated via `openmodelcontextprotocol.org/related-task` metadata. We recommend that a future SEP clarify the definition of "operation" in the progress specification.
    2. **Cancellation semantics**: With the notification-based approach, `notifications/cancelled` clearly targets the original request ID and causes the associated task to move to `cancelled` status, maintaining a clean separation between request cancellation and task lifecycle management.

While the notification is required by the specification for servers that create tasks, there are edge cases where it may be unavailable:

- **sHTTP without stream support**: In environments where either the client or the server does not support SSE streams, notifications cannot be delivered. In such cases, clients may choose to proactively poll with `tasks/get` using exponential backoff, though this is nonstandard and may result in unnecessary polling attempts if the server doesn't support tasks.
- **Degraded connection scenarios**: If the notification is lost in transit, clients should implement reasonable timeout behavior and fall back to the original response.

The standard and recommended approach is to wait for the `notifications/tasks/created` notification before beginning polling. Proactive polling without waiting for the notification should be considered a fallback mechanism for constrained environments only.

Design Decision: No Capabilities Declaration

Unlike other protocol features such as tools, resources, and prompts, tasks do not require capability negotiation. This decision was made to enable graceful degradation and per-request flexibility.

Task support can be determined implicitly through usage rather than explicitly through capability declarations. When a client sends a task-augmented request, the server will process it according to its capabilities. If the server doesn't support tasks for that request type, it simply ignores the task metadata and returns the result normally through the original request/response flow. The client can then detect the lack of task support by attempting to call `tasks/get` and handling any errors that result.

This approach eliminates the need for complex handshakes or feature detection protocols. Clients can optimistically try task augmentation and gracefully fall back to direct response handling if needed. This makes the protocol more resilient and easier to implement.

Additionally, this design provides per-request flexibility that would be difficult to express through capabilities. A server might support tasks on some request types but not others, or support might vary based on runtime conditions such as resource availability or load. Requiring granular capability declarations per request type would significantly complicate the protocol without providing substantial benefits. The implicit detection model is simpler and more flexible.

Alternative Designs Considered

Tool-Specific Async Execution: An earlier version of this proposal (#1391) focused specifically on tool calls, introducing an `invocationMode` field on tool definitions to mark tools as supporting synchronous, asynchronous, or both execution modes. This approach would have added dedicated fields to the tool call request and response structures, with server-side capability declarations to indicate support for async tool execution.

While this design would have addressed the immediate need for long-running tool calls, it was rejected in favor of the more general task primitive for several reasons. First, it artificially limited the async execution pattern to tools when other request types have similar needs. Resources can be expensive to read, prompts can require complex processing, and sampling requests may involve lengthy user interactions. Creating separate async patterns for each request type would lead to protocol fragmentation and inconsistent implementation patterns.

Second, the tool-specific approach required more complex capability negotiation and version handling. Servers would need to filter tool lists based on client capabilities, and SDKs would need to manage different invocation patterns for sync versus async tools. This complexity would ripple through every layer of the implementation stack.

Finally, the tool-specific design didn't address the broader architectural need for deferred result retrieval across all MCP request types. By generalizing to a task primitive that augments any request, this proposal provides a consistent pattern that can be applied uniformly across the protocol. More importantly, this foundation is extensible to future protocol messages and features such as subtasks, making it a more appropriate building block for the protocol's evolution.

Transport-Layer Solutions: An alternative approach would be to solve for this purely at the transport layer, without introducing a new data-layer primitive. Several proposals (#1335, #1442, #1597) address transport-specific concerns such as connection resilience, request retry semantics, and stream management for sHTTP. These are valuable improvements that can mitigate many scaling and reliability challenges associated with requests that may take extended time to complete.

However, transport-layer solutions alone are insufficient for the use cases this SEP addresses. Even with perfect transport-layer reliability, several data-layer concerns remain:

First, servers and clients need a way to communicate expectations about execution patterns. Without this, host applications cannot make informed decisions about UX patterns—should they block, show a spinner, or allow the user to continue working? An annotation alone could signal that a request might take extended time, but provides no mechanism to actively check status or retrieve results later.

Second, transport-layer solutions cannot provide visibility into the execution state of a request that is still in progress. If a request stops sending progress notifications, the client cannot distinguish between "the server is doing expensive work" and "the request was lost." Transport-level retries can confirm the connection is alive, but cannot answer "is this specific request still executing?" This visibility is critical for operations where users need confidence their work is progressing.

Third, different transports would require different mechanisms for these concerns. The sHTTP proposals adjust stream management and retry semantics to fulfill these requirements, but stdio has no equivalent extension points. This creates transport-specific fragmentation where implementers must solve the same problems

differently depending on their choice of transport. Data-layer operations provides consistent semantics across all transports.

Finally, deferred result retrieval and active status checks are data-layer concerns that cannot be addressed by transport improvements alone. The ability to retrieve a result multiple times, specify retention duration, and handle cleanup is orthogonal to how the underlying messages are delivered.

Resource-Based Approaches: Another possible approach would be to leverage existing MCP resources for tracking long-running operations. For example, a tool could return a linked resource that communicates operation status, and clients could subscribe to that resource to receive updates when the operation completes. This would allow servers to represent task state using the resource primitive, potentially with annotations for suggested polling frequency.

While this approach is technically feasible and servers remain free to adopt such conventions, it suffers from similar limitations as the tool-splitting pattern described in the Motivation section. Like the `start_tool` and `get_tool` convention, a resource-based tracking system would be convention-based rather than standardized, creating several challenges:

The most fundamental issue is the lack of a consistent way for clients to distinguish between ordinary resources (meant to be exposed to models) and status-tracking resources (meant to be polled by the application). Should a status resource be presented to the model? How should the client correlate a returned resource with the original tool call? Without standardization, different servers would implement different conventions, forcing clients/hosts/models to handle each server's particular approach. Extending resources with task-like semantics (such as polling frequency, keepalive durations, and explicit status states) would create a new and distinct purpose for resources that would be difficult to distinguish from their existing purpose as model-accessible content.

The resource subscription model has one additional issue: as it is push-based, it requires clients to wait for notifications of resource changes rather than actively polling for status. While this works for some use cases, it doesn't address scenarios where clients need to actively check status—for example, proactively and deterministically checking if work is still progressing, which is the original intent of this proposal.

The task primitive addresses these concerns by providing a standardized, protocol-level mechanism specifically designed for this use case, with consistent semantics that any client can leverage without host applications needing to understand server-specific conventions. While resource-based tracking remains possible for servers that prefer it and/or are already using it, this SEP provides a first-class alternative that solves the broader set of requirements identified previously.

Backward Compatibility

This SEP introduces **no backward incompatibilities**. All existing MCP functionality remains unchanged:

Compatibility Guarantees:

- Existing requests work identically with or without task metadata
- Servers that don't understand tasks process requests normally
- No protocol version negotiation required
- No capability declarations needed

Graceful Degradation:

- Clients race between waiting for the original request's response and waiting for the `notifications/tasks/created` notification followed by polling
- Whichever completes first (original response or task-based retrieval) is used by the client
- If a server doesn't support tasks, no `notifications/tasks/created` is sent, and the original request's response is used
- If a server supports tasks, the `notifications/tasks/created` notification is sent, enabling the client to begin polling for results

- This race pattern ensures graceful degradation without requiring capability negotiation or version detection
- Partial support is possible—servers can support tasks on some requests but not others

Adoption Path:

- Servers can implement task support incrementally, starting with high-value request types
- Clients can opportunistically use tasks where supported
- No coordination required between client and server updates

Future Work

The task primitive introduced in this SEP provides a foundation for several important extensions that will enhance MCP's workflow capabilities.

Push Notifications

While this SEP focuses on client-driven polling, future work could introduce server-initiated notifications for task state changes. This would be particularly valuable for operations that take hours or longer, where continuous polling becomes impractical.

A notification-based approach would allow servers to proactively inform clients when:

- A task completes or fails
- A task reaches a milestone or significant state transition
- A task requires input (complementing the `input_required` status)

This could be implemented through webhook-style mechanisms or persistent notification channels, depending on the transport capabilities. The proposed task ID and status model provides the necessary infrastructure for servers to identify which tasks warrant notifications and for clients to correlate notifications with their outstanding tasks.

Intermediate Results

The current task model returns results only upon completion. Future extensions could enable tasks to report intermediate results or progress artifacts during execution. This would support use cases where servers can produce partial outputs before final completion, such as:

- Streaming analysis results as they become available
- Reporting completed phases of multi-step operations
- Providing preview data while full processing continues

Intermediate results would build on the proposed task ID association mechanism, allowing servers to send multiple result notifications or response messages tied to the same task ID throughout its lifecycle.

Nested Task Execution

A significant future enhancement is support for hierarchical task relationships, where a task can spawn subtasks as part of its execution. This would enable complex, multi-step workflows orchestrated by the server.

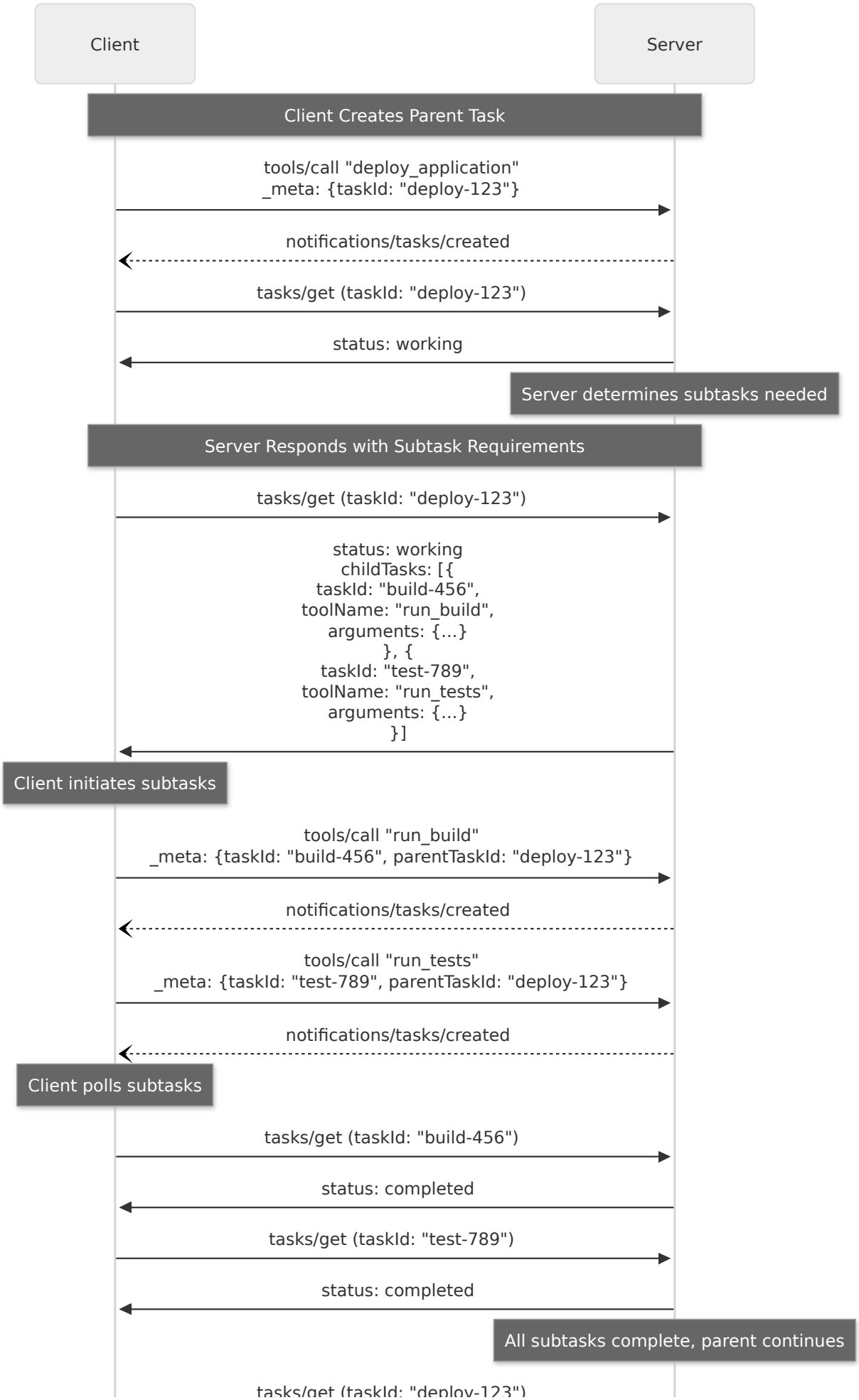
In a nested task model, a server could:

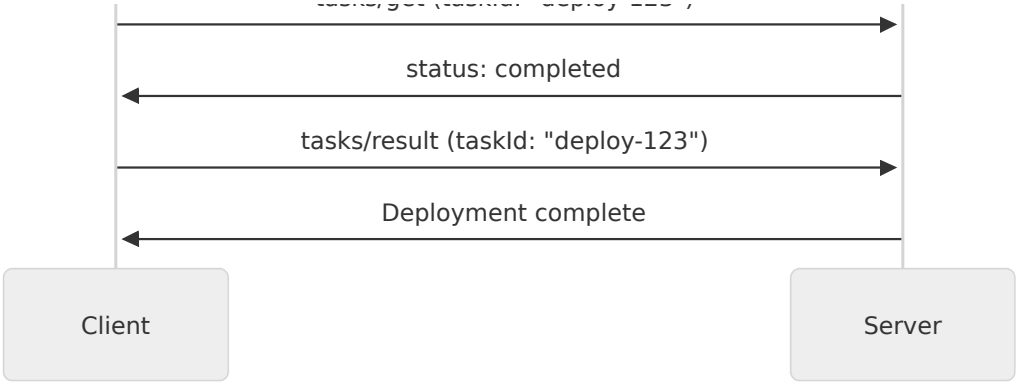
- Create subtasks in response to a parent task reaching a state that requires additional operations
- Communicate subtask requirements to the client, potentially including required tool calls or sampling requests
- Track subtask completion and use subtask results to advance the parent task
- Maintain provenance through task ID hierarchies, showing the relationship between parent and child tasks

For example, a complex analysis task might spawn several subtasks for data gathering, each represented by its own task ID but associated with the parent task. The parent task would remain in a pending state (potentially in a new `tool_required` status) until all required subtasks complete.

This hierarchical model would support sophisticated server-controlled workflows while maintaining the client's ability to monitor and retrieve results at any level of the task tree.

▼ Example nested task flow


Client
Server
Client Creates Parent Task
tools/call "deploy_application"
_meta: {taskId: "deploy-123"}
notifications/tasks/created
tasks/get (taskId: "deploy-123")
status: working
Server determines subtasks needed
Server Responds with Subtask Requirements
tasks/get (taskId: "deploy-123")
status: working
childTasks: [{
taskId: "build-456",
toolName: "run_build",
arguments: {...}
}, {
taskId: "test-789",
toolName: "run_tests",
arguments: {...}
}]
Client initiates subtasks
tools/call "run_build"
_meta: {taskId: "build-456", parentTaskId: "deploy-123"}
notifications/tasks/created
tools/call "run_tests"
_meta: {taskId: "test-789", parentTaskId: "deploy-123"}
notifications/tasks/created
Client polls subtasks
tasks/get (taskId: "build-456")
status: completed
tasks/get (taskId: "test-789")
status: completed
All subtasks complete, parent continues
tasks/get (taskId: "deploy-123")



Potential Data Model Extensions: The task status response could be extended to include parent and child task relationships:

```
{
  taskId: string;
  status: TaskStatus;
  keepAlive: number | null;
  pollFrequency?: number;
  error?: string;

  // Extensions for nested tasks
  parentTaskId?: string;        // ID of parent task, if this is a subtask
  childTasks?: Array<{          // Subtasks required by this task
    taskId: string;             // Pre-generated task ID for the subtask
    toolName: string;           // Tool to call for this subtask
    arguments?: object;         // Arguments for the tool call
  }>;
}
```

This would allow clients to:

- Discover subtasks required by a parent task through the `childTasks` array
- Initiate the required subtask tool calls using the pre-generated task IDs and provided arguments
- Navigate the task hierarchy by following parent/child relationships via `parentTaskId`
- Monitor all subtasks by polling each child task ID
- Wait for all subtasks to complete before checking parent task completion

The existing task metadata and status lifecycle are designed to be forward-compatible with these extensions.

SEP-1699 Support SSE polling via server-side disconnect

Final · Standards Track · Created 2025-10-22

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-10-22
- **Author(s)**: Jonathan Hefner (@jonathanhefner)
- **Issue**: #1699

Abstract

This SEP proposes changes to the Streamable HTTP transport in order to mitigate issues regarding long-running connections and resumability.

Motivation

The Streamable HTTP transport spec does not allow servers to close a connection while computing a result. In other words, barring client-side disconnection, servers must maintain potentially long-running connections.

Specification

When a server starts an SSE stream, it MUST immediately send an SSE event consisting of an `id` and an empty `data` string in order to prime the client to reconnect with that event ID as the `Last-Event-ID`.

Note that the SSE standard explicitly permits setting `data` to an empty string, and says that the appropriate client-side handling is to record the `id` for `Last-Event-ID` but otherwise ignore the event (i.e., not call the event handler callback).

At any point after the server has sent an event ID to the client, the server MAY disconnect at will. Specifically, this part of the MCP spec will be changed from:

> The server **SHOULD NOT** close the SSE stream before sending the JSON-RPC *response* for the received JSON-RPC *request*

To:

> The server **MAY** close the connection before sending the JSON-RPC *response* if it has sent an SSE event with an event ID to the client

If a server disconnects, the client will interpret the disconnection the same as a network failure, and will attempt to reconnect. In order to prevent clients from reconnecting / polling excessively, the server SHOULD send an SSE

event with a `retry` field indicating how long the client should wait before reconnecting. Clients MUST respect the `retry` field.

Rationale

Servers may disconnect at will, avoiding long-running connections. Sending a `retry` field will prevent the client from hammering the server with inappropriate reconnection attempts.

Backward Compatibility

- **New Client + Old Server**: No changes. No backward incompatibility.
- **Old Client + New Server**: Client should interpret an at-will disconnect the same as a network failure. `retry` field is part of the SSE standard. No backward incompatibility if client already implements proper SSE resuming logic.

Additional Information

This SEP supersedes (in part) SEP-1335.

SEP-1730 SDKs Tiering System

Final · Standards Track · Created 2025-10-29

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-10-29
- **Author(s)**: Inna Harper, Felix Weinberger
- **Issue**: #1730

Abstract

This SEP proposes a tiering system for Model Context Protocol (MCP) SDKs to establish clear expectations for feature support, maintenance commitments, and quality standards. The system defines three tiers of SDK support with objective, measurable criteria for classification.

Motivation

The MCP ecosystem needs SDK harmonization to help users make informed decisions. Users currently face challenges:

- **Feature Support Uncertainty**: No standardized way to know which SDKs support specific MCP features (OAuth, client/server/system features, like sampling, transports)
- **Maintenance Expectations**: Unclear commitment levels for bug fixes, security patches, and feature updates
- **Implementation Timelines**: No visibility into when SDKs will support new protocol versions and features

Specification

Tier Definitions

Tier 1: fully supported

SDKs in this tier provides full protocol implementation and is well supported

Requirements:

- **Feature complete and full support of the protocol**
  - All conformance tests pass
  - New protocol features before the new spec version release. (There is two week window between Release Candidate and the new protocol version release)
- **SDK maintenance**
  - Acknowledge and triage issues within two business days
  - Resolve security and critical bugs within seven days
  - Stable release and SDK versioning clearly documented
- **Documentation**

    - Comprehensive documentation with examples for all features
    - Published dependency update policy

Tier 2: commitment to be fully supported

SDKs with established implementations actively working toward full protocol support.

Requirements:

- **Feature complete and full support of the protocol**
    - 80% of conformance tests pass
    - New protocol features implemented within six months
- **SDK maintenance**
    - Active issue tracking and management
    - At least one stable release
- **Documentation**
    - Basic documentation covering core features
    - Published dependency update policy
- **Commitment to move to Tier1**
    - Published roadmap showing intent to achieve Tier 1 or, if SDK will remain in Tier 2 indefinitely, a transparent roadmap about the direction of the SDK and reasons for not being feature complete

Tier 3: Experimental

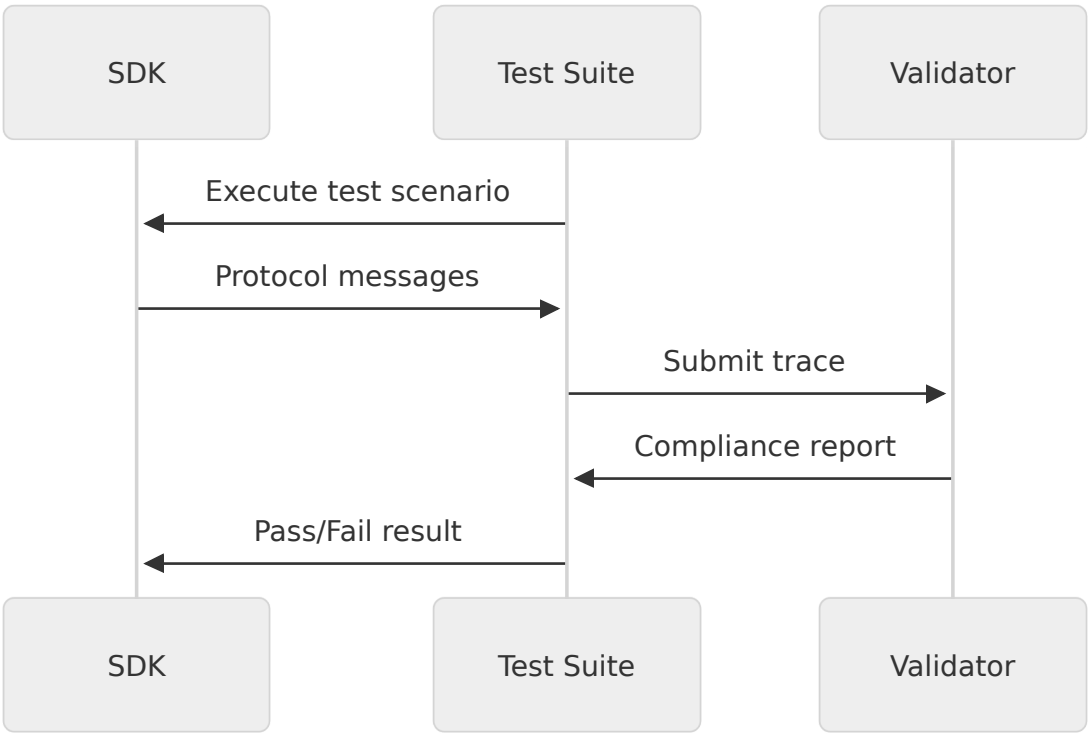
Early-stage or specialized SDKs exploring the protocol space.

Characteristics:

- No feature completeness guarantees
- No stable release requirement
- May focus on specific use cases or experimental features
- No timeline commitments for updates
- Suitable for niche implementations that may remain at this tier

Conformance Testing

All SDKs must undergo conformance testing using protocol trace validation: for details see Conformance Testing RFC (forthcoming). This SEP is not focusing on Conformance testing. For the initial version of tiering, we will go with the simplified version where we would have an Example server for each SDK and run simplified conformance tests against those.



Compliance Scoring:

- SDKs receive a percentage score based on test results
- Scores can be displayed as badges (e.g., "90% MCP Compliant")
- Tier 1: 100% compliance required
- Tier 2: 80% compliance required
- Tier 3: No minimum requirement

Tier Advancement Process

1. **Self-Assessment:** Maintainers evaluate their SDK against tier criteria
2. **Application:** Submit tier advancement request with evidence
3. **Review:** Community review period (2 weeks)
4. **Validation:** Automated conformance testing, github stats on issues
5. **Decision:** Tier assignment by MCP maintainers

Tier Relegation Process

1. **Auto validation:**
    1. compliance tests continuously not passing for four week for Tier 1
    2. 20% of compliance tests continuously not passing for four week for Tier 2
2. Issues:
    1. Issues are not addressed within two months

Requirements matrix

Feature	SDK A	SDK B	SDK C
Protocol Features support (Conformance tests)	85%	60%%	100%
GitHub support stats	10 days	100 days	5 days
Documentation (self reported)	Good	Minimal	Good
Tier (computed from above)	Tier 2	Tier 3	Tier 1

Rationale

Why Three Tiers?

- **Tier 1** ensures users have well supported, fully-featured SDK
- **Tier 2** provides a clear pathway for improving SDKs
- **Tier 3** allows experimentation without creating barriers to entry

Why Time-Based Commitments?

While the community raised concerns about rigid timelines, they provide:

- Clear expectations for users
- Measurable goals for maintainers
- Flexibility through tier progression

Why Not Just Feature Matrices?

Feature matrices alone don't communicate:

- Maintenance commitment
- Quality standards
- Support expectations

The tiering system combines feature support with quality guarantees.

Alternatives Considered

1. Feature Matrix Only

Rejected because: Doesn't communicate maintenance commitments or quality standards

2. Percentage-Based Scoring

Rejected because: Too granular and doesn't capture qualitative aspects like support

3. Properties-Based System

Rejected because: Multiple overlapping properties could confuse users

4. Latest Version Listing Only

Rejected because: Simply listing "supports MCP date" fails to capture critical information:

- Version support may be incomplete (e.g., supports <date> except OAuth)
- No indication of maintenance commitment or issue response times
- Lacks information about security patch timelines
- Doesn't communicate dependency update policies
- Version numbers alone don't indicate production readiness

5. No Formal System

Rejected because: Current ad-hoc approach creates uncertainty for users

Backward Compatibility

This proposal introduces a new classification system with no breaking changes:

- Existing SDKs continue to function
- Classification is opt-in initially
- Grace period for existing SDKs to achieve tier status

Security Implications

- Tier 1 SDKs must address security issues within 7 days
- All tiers encouraged to follow security best practices
- Conformance tests include security validation

Implementation Plan

- [ ] Finalize simplified conformance test suite - Nov 4, 2025
- [ ] SDK maintainers self-assess and apply for tiers - Nov 14, 2025
- [ ] Initial tier assignments - before the November spec release
- [ ] Implement full compliance tests
- [ ] Implement automatic issue tracking analysis for SDKs

Community Impact

SDK Maintainers

- Clear goals for improvement
- Recognition for quality implementations
- Structured pathway for advancement

SDK Users

- Informed selection of SDKs
- Clear expectations for support
- Confidence in tier 1 implementations

Ecosystem

- Improved overall SDK quality
- Standardized feature support
- Healthy competition between implementations

References

- SDK Maintainer Meeting Notes (#1648)
- SDK Harmonization Goals (#1444)
- Conformance Testing SEP (DRAFT)

Appendix

Simplified conformance tests

While we are working on a comprehensive proposal for conformance testing which will take some time to implement, we want to move forward with at least some automated way to check if SDK has a full set of features. We will start from Servers features set, as we have many more servers than clients and the vast majority of developers using SDKs are Server implementers.

The most straightforward approach is to have an Example Server for each SDK, similar to to Everything Server. Then we will have Conformance Test Client with all the test cases we want to be able to test, for example:

- execute “hello world” tool
- Get prompt
- Get completion
- Get resource template
- Receive notifications

What is needed form SDKs maintainers: implement everything server based on a spec. Spec will look like:

- Tool “say_hello” to return simple text
- Tool “show_image” to return and image
- Tool “tool_with_logging” to return structured output in a format <> and log three events: start, process, end
- Tool "tool_with_notifications" to return structured output in a format <> and have two notifications <>

Given well defined spec for the server and SDK documentation, it should be easy to implement it with the help of any coding agent. We want to check it into each SDKs repo as it will serve as an example for server implementers.

Once each SDK has an Everything server, we will run the Conformance Test Client against it.

SEP-1850 PR-Based SEP Workflow

Final · Process · Created 2025-11-20

- **Status**: Final
- **Type**: Process
- **Created**: 2025-11-20
- **Accepted**: 2025-11-28, 8 Yes, 0 No, 0 Absent per vote in Discord.
- **Author(s)**: Nick Cooper (@nickcoai), David Soria Parra (@davidsp)
- **Sponsor**: David Soria Parra (@davidsp)
- **PR**: https://github.com/modelcontextprotocol/specification/pull/1850

Abstract

This SEP formalizes the pull request-based SEP workflow that stores proposals as markdown files in the `seps/` directory of the Model Context Protocol specification repository. The workflow assigns SEP numbers from pull request numbers, maintains version history in Git, and replaces the previous GitHub Issues-based process. This establishes a file-based approach as the canonical way to author, review, and accept SEPs.

Motivation

The issue-based SEP process introduced several challenges:

- **Dispersed content**: Proposal content was scattered across GitHub issues, linked documents, and pull requests, making review and archival difficult.
- **Difficult collaboration**: Maintaining long-form specifications in issue bodies made iterative edits and multi-contributor collaboration harder.
- **Limited version control**: GitHub issues don't provide the same version control capabilities as Git-managed files.
- **Unclear status management**: The process lacked clear mechanisms for tracking status transitions and ensuring consistency between different sources of truth.

A file-based workflow addresses these issues by:

- Keeping every SEP in version control alongside the specification itself
- Providing Git's built-in review tooling, history, and searchability
- Linking SEP numbers to pull requests to eliminate manual bookkeeping
- Surfacing all discussion in the pull request thread
- Using PR labels in conjunction with file status for better discoverability

Specification

1. Canonical Location

- Every SEP lives in `seps/{NUMBER}-{slug}.md` in the specification repository
- The SEP number is always the pull request number that introduces the SEP file
- The `seps/` directory serves as the single source of truth for all SEPs

2. Author Workflow

1. **Draft the proposal** in `seps/0000-{slug}.md` using `0000` as a placeholder number
2. **Open a pull request** containing the draft SEP and any supporting materials
3. **Request a sponsor** from the Maintainers list; tag potential sponsors from MAINTAINERS.md
4. **After the PR number is known**, amend the commit to rename the file to `{PR-number}-{slug}.md` and update the header (`SEP-{PR-number}` and `PR: #{PR-number}`)
5. **Wait for sponsor assignment**: Once a sponsor agrees, they will assign themselves and update the status to `Draft`

3. Sponsor Responsibilities

A Sponsor is a Core Maintainer or Maintainer who champions the SEP through the review process. The sponsor's responsibilities include:

- **Reviewing the proposal** and providing constructive feedback
- **Requesting changes** based on community input
- **Managing status transitions** by:
  - Ensuring that the `Status` field in the SEP markdown file is accurate
  - Applying matching PR labels to keep them in sync with the file status
  - Communicating status changes via PR comments
- **Initiating formal review** when the SEP is ready (moving from `Draft` to `In-Review`)
- **Raising to Core-Maintainers** ensuring the SEP is presented at the Core Maintainer meeting and that author and sponsor present.
- **Ensuring quality standards** are met before advancing the proposal
- **Tracking implementation** progress and ensuring reference implementations are complete before `Final` status

4. Review Flow

Status progression follows: `Draft → In-Review → Accepted → Final`

Additional terminal states: `Rejected` , `Withdrawn` , `Superseded` , `Dormant`

Dormant status: If a SEP does not find a sponsor within six months, Core Maintainers may close the PR and mark the SEP as `dormant` .

Reference implementations must be tracked via linked pull requests or issues and must be complete before marking a SEP as `Final` .

5. Documentation

- `docs/community/sep-guidelines.mdx` serves as the contributor-facing instructions
- `seps/README.md` provides the concise reference for formatting, naming, sponsor responsibilities, and acceptance criteria
- Both documents must reflect this workflow and be kept in sync

6. SEP File Structure

Each SEP must include:

```
# SEP-{NUMBER}: {Title}

- **Status**: Draft | In-Review | Accepted | Rejected | Withdrawn | Final | Superseded |
Dormant
- **Type**: Standards Track | Informational | Process
- **Created**: YYYY-MM-DD
- **Author(s)**: Name <email> (@github-username)
- **Sponsor**: @github-username (or "None" if seeking sponsor)
- **PR**: https://github.com/modelcontextprotocol/specification/pull/{NUMBER}

## Abstract

## Motivation

## Specification

## Rationale

## Backward Compatibility

## Security Implications

## Reference Implementation
```

7. Status Management via PR Labels

To improve discoverability and filtering:

- Sponsors must apply PR labels that match the SEP status (`draft`, `in-review`, `accepted`, `final`, etc.)
- Both the markdown `Status` field and PR labels should be kept in sync
- The markdown file serves as the canonical record (versioned with the proposal)
- PR labels enable easy filtering and searching for SEPs by status
- Only sponsors should modify status fields and labels; authors should request changes through their sponsor

8. Legacy Considerations

- Contributors may optionally open a GitHub Issue for early discussion, but the authoritative SEP text lives in `seps/`
- Issues should link to the relevant file once a pull request exists
- SEP numbers are derived from PR numbers, not issue numbers

Rationale

Why File-Based?

Storing SEPs as files keeps authoritative specs versioned with the code, mirroring successful processes used by PEPs (Python Enhancement Proposals) and other standards bodies. This approach:

- Provides built-in version control via Git
- Enables standard code review workflows

- Maintains clear history of all changes
- Supports multi-contributor collaboration
- Integrates naturally with the specification repository

Why PR Numbers?

Using pull request numbers:

- Eliminates race conditions around manual numbering
- Creates natural traceability between proposal and discussion
- Prevents number conflicts
- Simplifies the contribution process
- Maintains a single discussion thread for review

Why PR Labels?

Adding PR labels alongside the file status:

- Enables quick filtering of SEPs by status without opening files
- Provides immediate visibility of SEP states in PR lists
- Supports GitHub's search and filter capabilities
- Complements the canonical markdown status field
- Reduces friction for maintainers managing multiple SEPs

Making This the Primary Process

Maintaining two overlapping canonical processes risked divergence and created confusion for contributors. Establishing the file-based approach as the primary method:

- Reduces cognitive overhead for new contributors
- Ensures consistency in the SEP corpus
- Simplifies maintenance for sponsors
- Aligns with industry best practices

Backward Compatibility

- Existing issue-based SEPs remain valid and require no migration
- Historical GitHub Issue links continue to work
- Future SEPs should reference the new file locations in `seps/`
- Maintainers may optionally backfill historical SEPs into `seps/` for archival purposes

Security Implications

No new security considerations beyond the standard code review process for pull requests.

Reference Implementation

- This pull request (#1850) implements the canonical instructions in both `seps/README.md` and `docs/community/sep-guidelines.mdx`
- The process has been updated to reflect the PR-based workflow with status management via labels
- This SEP document itself serves as an example of the new format

Vote

This SEP was accepted unanimously by the MCP Core Maintainers with a vote of 8 yes's, 0 no's and 0 absent votes on Friday December 28th, 2025 in a Discord poll.

SEP-1865 MCP Apps - Interactive User Interfaces for MCP

Final · Extensions Track · Created 2025-11-21

- **Status**: Final
- **Type**: Extensions Track
- **Created**: 2025-11-21
- **Author(s)**: Ido Salomon (@idosal), Liad Yosef (@liadyosef), Olivier Chafik (@olivierchafik), Jerome Swannack (@jeromeswannack), Jonathan Hefner (@jonathanhefner), Anton Pidkuiko (@antonpidkuiko), Nick Cooper (@nickcooper), Bryan Ashley (@bryanashley), Alexi Christakis (@alexichristakis)
- **Sponsor**: None (seeking sponsor)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/1865

The full extension specification is maintained in the [ext-apps repository] (https://github.com/modelcontextprotocol/ext-apps/tree/main/specification).

Abstract

This SEP proposes an extension to MCP (per SEP-1724) that enables servers to deliver interactive user interfaces to hosts. MCP Apps introduces a standardized pattern for declaring UI resources via the `ui://` URI scheme, associating them with tools through metadata, and facilitating bi-directional communication between the UI and the host using MCP's JSON-RPC base protocol. This extension addresses the growing community need for rich, interactive experiences in MCP-enabled applications, maintaining security, auditability, and alignment with MCP's core architecture. The initial specification focuses on HTML resources (`text/html;profile=mcp-app`) with a clear path for future extensions.

Motivation

MCP lacks a standardized way for servers to deliver rich, interactive user interfaces to hosts. This gap blocks many use cases that require visual presentation and interactivity that go beyond plain text or structured data. As more hosts adopt this capability, the risk of fragmentation and interoperability challenges grows.

MCP-UI has demonstrated the viability and value of MCP apps built on UI resources and serves as a community playground for the UI spec and SDK. Fueled by a dedicated community, it developed the bi-directional communication model and the HTML, external URL, and remote DOM content types. MCP-UI's adopters, including hosts and providers such as Postman, HuggingFace, Shopify, Goose, and ElevenLabs, have provided critical insights and contributions to the community.

OpenAI's Apps SDK, launched in November 2025, further validated the demand for rich UI experiences within conversational AI interfaces. The Apps SDK enables developers to build rich, interactive applications inside ChatGPT using MCP as its backbone.

The architecture of both the Apps SDK and MCP-UI has significantly informed the design of this specification.

However, without formal standardization:

- Servers cannot reliably expect UI support via MCP
- Each host may implement slightly different behaviors

- Security and auditability patterns are inconsistent
- Developers must maintain separate implementations or adapters for different hosts (e.g., MCP-UI vs. Apps SDK)

This SEP addresses the current limitations through an optional, backwards-compatible extension that unifies the approaches pioneered by MCP-UI and the Apps SDK into a single, open standard.

Specification

The full specification can be found at modelcontextprotocol/ext-apps.

At a high level, MCP Apps extends the Model Context Protocol to enable servers to deliver interactive user interfaces to hosts. This extension introduces:

- **UI Resources:** Predeclared resources using the `ui://` URI scheme
- **Resource Discovery:** Tools reference UI resources via metadata
- **Bi-directional Communication:** UI iframes communicate with hosts using standard MCP JSON-RPC protocol
- **Security Model:** Mandatory iframe sandboxing with auditable communication

This specification focuses on HTML content (`text/html;profile=mcp-app`) as the initial content type, with extensibility for future formats.

As an extension, MCP Apps is optional and must be explicitly negotiated between clients and servers through the extension capabilities mechanism (see Capability Negotiation section in the full specification).

Rationale

Predeclared resources vs. inline embedding

UI is modeled as predeclared resources (`ui://`), referenced by tools via metadata. This allows:

- Hosts to prefetch templates before tool execution, improving performance
- Separation of presentation (template) from data (tool results), facilitating caching
- Security review of UI resources

Alternatives considered:

- **Embedded resources:** Current MCP-UI approach, where resources are returned in tool results. Although it's more convenient for server development, it was deferred due to the gaps in performance optimization and the challenges in the UI review process.
- **Resource links:** Predeclare the resources but return links in tool results. Deferred due to the gaps in performance optimization.

Reusing MCP JSON-RPC instead of a custom protocol

Reuses existing MCP infrastructure (type definitions, SDKs, etc.). JSON-RPC offers advanced capabilities (timeouts, errors, etc.).

Alternatives considered:

- **Custom message protocol:** Current MCP-UI approach with message types like tool, intent, prompt, etc. These message types can be translated to a subset of the proposed JSON-RPC messages.
- **Global API object:** Rejected because it requires host-specific injection and doesn't work with external iframe sources. Syntactic sugar may still be added on the server/UI side.

HTML-only MVP

- HTML is universally supported and well-understood
- Simplest security model (standard iframe sandbox)
- Allows screenshot/preview generation (e.g., via html2canvas)
- Sufficient for most observed use cases
- Provides a clear baseline for future extensions

Alternatives considered:

- **Include external URLs in MVP:** This is one of the easiest content types for servers to adopt, as it's possible to embed regular apps. However, it was deferred due to concerns around model visibility, inability to screenshot content, and review process. It may effectively be supported with the SEP's new `externalIframes` capability.

Backward Compatibility

The proposal is an optional extension to the core protocol. Existing implementations continue working without changes.

Security Implications

Hosting interactive UI content from potentially untrusted MCP servers requires careful security consideration.

Based on the threat model, MCP Apps proposes the following mitigations:

- **Iframe sandboxing**: All UI content runs in sandboxed iframes with restricted permissions
- **Predeclared templates**: Hosts can review HTML content before rendering
- **Auditable messages**: All UI-to-host communication goes through loggable JSON-RPC
- **User consent**: Hosts can require explicit approval for UI-initiated tool calls

A full threat model analysis and mitigations are available in the full specification.

Reference Implementation

- MCP-UI client and server SDKs support the patterns proposed in this spec.
- ext-apps repository contains a prototype implementation by Olivier Chafik.

SEP-2085 Governance Succession and Amendment Procedures

Final · Process · Created 2025-12-05

- **Status**: Final
- **Type**: Process
- **Created**: 2025-12-05
- **Author(s)**: David Soria Parra (@dsp-ant)
- **Sponsor**: David Soria Parra (@dsp-ant)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2085

Abstract

This SEP establishes formal procedures for Lead Maintainer succession and governance amendment within the Model Context Protocol project. It defines clear processes for leadership transitions when a Lead Maintainer leaves their role and establishes requirements for proposing and approving changes to the governance structure itself.

Motivation

The current MCP governance structure defines roles and responsibilities but lacks explicit procedures for two critical scenarios:

1. **Leadership Succession**: The governance document identifies Justin Spahr-Summers and David Soria Parra as Lead Maintainers (BDFLs) but does not specify what happens if one or both leave their roles. Without a defined succession process, an unexpected departure could create uncertainty about project leadership and decision-making authority.
2. **Governance Evolution**: As the MCP project grows and the community evolves, the governance structure may need to adapt. Currently, there is no defined process for how the governance document itself can be amended, which could lead to ad-hoc changes without proper community input or unclear authority for making such changes.

Establishing these procedures now, while the project leadership is stable, ensures continuity and provides clear guidance for future scenarios.

Specification

The following sections shall be added to the MCP Governance document.

Succession

If a Lead Maintainer leaves their role for any reason, the succession process begins upon their written notice or, if unable to provide notice, upon a determination by the remaining Lead Maintainer(s) or Core Maintainers that the Lead Maintainer is unable to continue serving.

If one or more Lead Maintainer(s) remain, they shall appoint a successor (by majority vote if multiple), and the remaining Lead Maintainer(s) will continue to govern until a successor is appointed.

If no Lead Maintainers remain, the Core Maintainers shall appoint a successor by majority vote within 30 days, and the project operates by two-thirds vote of Core Maintainers until a new Lead Maintainer is appointed.

Amendment

Amendments to this governance structure may only be proposed by Lead Maintainers. Any proposed amendment must be approved by a two-thirds (2/3) majority of all Core Maintainers to take effect.

Amendment proposals shall:

1. Be submitted in writing with clear rationale for the proposed change
2. Include specific language describing the modification to existing governance provisions
3. Allow for a minimum comment period of five (5) days before voting
4. Be decided by recorded vote of Core Maintainers

Rationale

Succession Process Design

The succession process is designed with several principles in mind:

- **Continuity**: Remaining Lead Maintainers can continue operating and appoint successors without disruption to project governance.
- **Fallback Authority**: If all Lead Maintainers depart, Core Maintainers have clear authority to select new leadership, preventing a governance vacuum.
- **Time-Bound Process**: The 30-day requirement ensures succession happens promptly while allowing adequate time for deliberation.
- **Supermajority Interim Governance**: Two-thirds voting during interregnum periods ensures major decisions have broad support during transitional periods.

Amendment Process Design

The amendment process balances stability with adaptability:

- **Lead Maintainer Proposal Authority**: Limiting proposal authority to Lead Maintainers prevents governance churn from frequent amendment proposals while ensuring those with deepest project investment can drive necessary changes.
- **Core Maintainer Approval**: Requiring two-thirds Core Maintainer approval ensures amendments have broad support from those actively governing the project.
- **Comment Period**: The five-day minimum comment period allows affected parties to review and provide input before voting.
- **Recorded Votes**: Transparency in voting ensures accountability and provides a historical record of governance decisions.

Alternatives Considered

Succession by Election: An open election process was considered but rejected as potentially disruptive and slow during critical transition periods. The current proposal allows for quick succession while maintaining checks through the existing maintainer structure.

Amendment by Any Maintainer: Allowing any maintainer to propose amendments was considered but could lead to governance instability. The current approach balances stability with the ability to evolve.

Longer Comment Periods: Longer comment periods (e.g., 30 days) were considered but deemed excessive for a project that already has regular bi-weekly Core Maintainer meetings. Five days allows for at least one meeting cycle while enabling timely decisions.

Backward Compatibility

This SEP adds new procedures without modifying existing governance structures. No backward compatibility concerns exist.

Security Implications

This SEP has no direct security implications. However, clear succession procedures indirectly support security by ensuring continuous responsible stewardship of the project, including security-related decisions.

Reference Implementation

Upon acceptance, this SEP will be implemented by adding the Succession and Amendment sections to `docs/community/governance.mdx`. The new sections will be inserted after the "Lead Maintainers (BDFL)" section and before the "Decision Process" section.

A draft pull request implementing these changes will be linked here once available.

SEP-2106 Tools `inputSchema` & `outputSchema` Conform to JSON Schema 2020-12

Final · Standards Track · Created 2026-01-06

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-01-06
- **Author(s)**: John McBride (@jpmcb) — original proposal; Ola Hungerford (@olaservo) — current shepherd, post-SEP-1850 conversion
- **Sponsor**: Ola Hungerford (@olaservo)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2106

> **Authorship note:** The original proposal was authored by John McBride (@jpmcb) in PR #881, prior to the SEP-1850 PR-based workflow. This file converts that proposal to the current SEP format and is shepherded by Ola Hungerford (@olaservo), who has also revised the Backward Compatibility, Security Implications, and SDK Migration sections in response to review feedback. The original prose and design intent remain John's; substantive changes since the conversion are tracked in this PR's commit history.

Abstract

This SEP proposes loosening the restrictions on `inputSchema`, `outputSchema`, and `structuredContent` to better support JSON Schema 2020-12. Specifically:

- **`inputSchema`**: Keeps `type: "object"` required (since tool arguments are objects), but allows any additional JSON Schema properties to support powerful validation compositions (`anyOf`, `oneOf`, `allOf`, etc.)
- **`outputSchema`**: Fully supports JSON Schema 2020-12 since MCP servers may return any valid JSON
- **`structuredContent`**: Accepts any JSON value validated by `outputSchema`

This proposal enables MCP servers to leverage the expressiveness of JSON Schema 2020-12 while maintaining backward compatibility with existing implementations.

Motivation

The current MCP specification restricts tool schemas in ways that conflict with full JSON Schema support:

1. **inputSchema restriction**: Currently only allows `type`, `properties`, and `required` fields. This prevents use of composition keywords like `anyOf`, `oneOf`, and `allOf` for sophisticated object validation patterns.
2. **outputSchema restriction**: Also restricted to `type: "object"` with only `properties` and `required`, despite the specification claiming to support "JSON Schema."
3. **structuredContent restriction**: Defined as `{ [key: string]: unknown }` (an object with string keys), which prevents returning arrays—a common API response pattern.

Real-World Impact

Consider a weather API tool that returns hourly forecasts:

```
[
  { "hour": "09:00", "temp": 68, "conditions": "sunny" },
  { "hour": "10:00", "temp": 72, "conditions": "partly cloudy" },
  { "hour": "11:00", "temp": 75, "conditions": "cloudy" }
]
```

Currently, this natural array response is **impossible** because `structuredContent` must be an object. Developers are forced to wrap arrays in unnecessary container objects:

```
{
  "forecasts": [
    { "hour": "09:00", "temp": 68, "conditions": "sunny" },
    ...
  ]
}
```

This artificial constraint:

- Adds unnecessary nesting to responses
- Conflicts with common REST API patterns
- Prevents direct schema validation of array responses

Schema Composition Use Cases

The current `inputSchema` restriction prevents legitimate schema patterns. With this SEP, tools can use composition keywords alongside `type: "object"`:

```
{
  "type": "object",
  "oneOf": [
    { "properties": { "id": { "type": "string" } }, "required": ["id"] },
    { "properties": { "name": { "type": "string" } }, "required": ["name"] }
  ]
}
```

This pattern allows a tool to accept either an ID-based or name-based lookup—a common API design that is currently unsupported because the schema only allows `type`, `properties`, and `required` fields.

Specification

1. Loosen inputSchema

Current definition:

```
inputSchema: {
  type: "object";
  properties?: { [key: string]: object };
  required?: string[];
};
```

Proposed definition:

```
inputSchema: {
  $schema?: string;
  type: "object";
  [key: string]: unknown;
};
```

The `inputSchema` field retains the `type: "object"` requirement (since tool arguments are always objects), but now accepts any additional JSON Schema properties. This enables:

- Composition keywords: `anyOf`, `oneOf`, `allOf`, `not`
- Conditional schemas: `if` / `then` / `else`
- Reference schemas: `$ref`, `$defs`
- Any other valid JSON Schema 2020-12 keywords

2. Loosen outputSchema

Current definition:

```
outputSchema?: {
  type: "object";
  properties?: { [key: string]: object };
  required?: string[];
};
```

Proposed definition:

```
outputSchema?: {
  $schema?: string;
  [key: string]: unknown;
};
```

The `outputSchema` field accepts any valid JSON Schema 2020-12 object, enabling schemas that validate arrays, primitives, or complex compositions. Unlike `inputSchema`, there is no `type: "object"` requirement since tool outputs can be any valid JSON.

3. Loosen structuredContent

Current definition:

```
structuredContent?: { [key: string]: unknown };
```

Proposed definition:

```
structuredContent?: unknown;
```

The `structuredContent` field accepts any valid JSON value that conforms to the tool's `outputSchema`. This includes:

- Objects: `{ "key": "value" }`
- Arrays: `[1, 2, 3]` or `[{ "id": "abc" }, { "id": "xyz" }]`
- Primitives: `"string"`, `42`, `true`, `null`

4. Documentation Updates

Update `docs/specification/draft/server/tools.mdx`:

- Remove statement that `structuredContent` is "returned as a JSON object"
- Clarify that `structuredContent` can be any JSON value conforming to `outputSchema`
- Add examples demonstrating array responses

5. Examples

Tool returning an array of objects:

```
{
  "name": "list_users",
  "description": "List all users in the system",
  "inputSchema": {
    "type": "object",
    "properties": {
      "limit": { "type": "integer", "minimum": 1, "maximum": 100 }
    }
  },
  "outputSchema": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "id": { "type": "string" },
        "name": { "type": "string" },
        "email": { "type": "string", "format": "email" }
      },
      "required": ["id", "name"]
    }
  }
}
```

Response:

```
{
  "content": [
    {
      "type": "text",
      "text": "Found 2 users: Alice (u1, alice@example.com) and Bob (u2,
bob@example.com)."
    }
  ],
  "structuredContent": [
    { "id": "u1", "name": "Alice", "email": "alice@example.com" },
    { "id": "u2", "name": "Bob", "email": "bob@example.com" }
  ]
}
```

Tool with composition schema:

```
{
  "name": "find_resource",
  "description": "Find a resource by ID or name",
  "inputSchema": {
    "type": "object",
    "oneOf": [
      {
        "properties": { "id": { "type": "string", "format": "uuid" } },
        "required": ["id"]
      },
      {
        "properties": { "name": { "type": "string", "minLength": 1 } },
        "required": ["name"]
      }
    ]
  }
}
```

Rationale

Why not just allow arrays?

While we could simply extend `structuredContent` to allow arrays, this would be an incomplete solution. The root cause is that the schema types are artificially restricted to `type: "object"`. By allowing any valid JSON Schema, we:

1. Enable the full power of JSON Schema 2020-12
2. Align with the specification's claim of JSON Schema support
3. Provide a consistent, principled approach rather than piecemeal fixes

Why not require a wrapper object?

Requiring arrays to be wrapped in objects (e.g., `{ "items": [...] }`) was considered but rejected because:

1. It adds unnecessary complexity to responses
2. It conflicts with common API design patterns
3. It prevents direct schema validation of the actual response structure
4. JSON Schema already handles array validation elegantly

Real-World API Patterns

Many production APIs return arrays directly:

- **GitHub Events API**: Returns arrays of event objects
- **AccuWeather Search API**: Returns arrays of location matches
- **REST collection endpoints**: Standard `GET /users` returns `[{...}, {...}]`

Forcing wrapper objects creates friction for developers integrating existing APIs with MCP. Generic JSON Schema validation libraries should work without MCP-specific customization.

Alignment with JSON Schema 2020-12

JSON Schema 2020-12 provides powerful features for schema composition and validation. By removing artificial restrictions, MCP aligns with industry standards (OpenAPI 3.1 uses JSON Schema 2020-12) and enables developers to leverage existing JSON Schema knowledge and tooling.

SDK Ecosystem Evidence

The friction caused by current restrictions is not theoretical. FastMCP, one of the most popular Python SDKs for MCP, has implemented extensive workarounds:

1. **Explicit error messages** acknowledge the limitation:

   ```
   raise ValueError(
       f"Output schemas must represent object types due to MCP spec limitations."
   )
   ```

2. **Auto-wrapping infrastructure** adds complexity:
   - A `_WrappedResult` dataclass wraps non-object returns
   - A custom `x-fastmcp-wrap-result` extension enables client-side unwrapping
   - Both SDK and client need matching wrap/unwrap logic

3. **Real bugs** have resulted from these workarounds:
   - Issue #2455: `$ref` schemas without `type: object` broke ALL tools on the server
   - Issue #2421: Unexpected `{"result": ...}` wrapping confused users

This demonstrates that the current restrictions create genuine ecosystem friction that SEP-2106 would eliminate.

OpenAPI Precedent

The OpenAPI specification went through a similar evolution. OpenAPI 3.0 used an "extended subset" of JSON Schema with custom restrictions (like requiring `nullable: true` instead of allowing `"null"` as a type).

OpenAPI 3.1 made the strategic decision to fully align with JSON Schema 2020-12, accepting breaking changes to eliminate the friction. The result: better tooling compatibility and less ecosystem confusion.

OpenAPI's Problem	MCP's Parallel
`type` must be string, not array	`inputSchema` only allows specific fields
Couldn't use standard null handling	Can't use `oneOf` / `anyOf` in schemas
Custom `nullable` keyword	Object-only `structuredContent`
Caused tooling confusion	Causes SDK workarounds

MCP can learn from OpenAPI's experience rather than repeating the same evolution over several years.

Backward Compatibility

This change is **wire-format backward compatible** but has nuances depending on the direction of the version mismatch.

Compatibility Matrix

	New client (post-SEP)	Old client (pre-SEP)
New server (post-SEP)	Fully compatible.	Compatible **only when the server returns object-typed `structuredContent`**. Arrays/primitives in `structuredContent` may break.
Old server (pre-SEP)	Fully compatible. Existing object-only schemas remain valid.	Unchanged.

The asymmetry: a new server that takes advantage of array or primitive `structuredContent` (or composition keywords in `inputSchema`) cannot assume an old client will accept the response. Old clients written against the previous wire format may reject `structuredContent` that is not a JSON object, or fail to validate `inputSchema` containing keywords beyond `type` / `properties` / `required`.

To remain interoperable with older clients, **servers using array or primitive `structuredContent` MUST also emit a `TextContent` block containing the serialized JSON** (as already recommended in the tools specification). Clients that do not understand non-object `structuredContent` can fall back to the text content.

TypeScript / SDK Migration

Widening the `structuredContent` field type from `{ [key: string]: unknown }` to `unknown` is a **source-breaking change for typed consumers**, even though the wire format is unchanged. Code such as:

```
const result = await client.callTool({ name: "get_weather", arguments: { ... } });
const temp = result.structuredContent?.temperature;        // previously compiled (type: unknown)
const city = result.structuredContent?.["city"] as string; // previously compiled
```

will no longer type-check after the change, because TypeScript forbids property access on `unknown` without a narrowing guard:

```
const sc = result.structuredContent;
if (sc && typeof sc === "object" && !Array.isArray(sc)) {
  const temp = (sc as Record<string, unknown>).temperature;
}
```

This break is intentional — the previous type was a lie whenever a tool returned a non-object — but SDK maintainers SHOULD:

- Document the migration in SDK release notes.
- Where ergonomic, provide typed helpers (e.g. generics over a tool's `outputSchema`) so consumers do not need to write narrowing guards by hand.

Migration Path

- **Servers**: No migration is required to keep working as before. To use array or primitive `structuredContent`, also emit a serialized `TextContent` fallback.
- **Clients**: Old clients continue to work against object-only servers. To consume the new flexibility, accept any JSON value in `structuredContent` and validate against `outputSchema` if present.
- **SDKs**: Update generated types to mirror the new schema (`unknown` for `structuredContent`, open-ended `inputSchema` / `outputSchema`) and call out the source-breaking type change in release notes.

Security Implications

JSON Schema validation already handles type checking, value constraints, and required field validation, and implementations MUST continue to validate all inputs and outputs against declared schemas. Allowing the full JSON Schema 2020-12 vocabulary surfaces two areas that warrant explicit guidance.

`$ref` Dereferencing (SSRF and Fetch-DoS)

JSON Schema 2020-12 permits `$ref` to point at an absolute URI, not just a JSON Pointer into the same document. A naive implementation that resolves every `$ref` it encounters by issuing an HTTP request gives an attacker a server-side request forgery / fetch amplification primitive: a malicious tool definition can cause the host to fetch arbitrary URLs, including internal metadata endpoints or large payloads designed to exhaust resources.

To mitigate this:

- Implementations **MUST NOT** automatically dereference `$ref` values that resolve to a network URI (i.e. anything that is not a same-document JSON Pointer such as `#/$defs/Foo` or an internal `$anchor`).
- "Automatically" here means "as part of normal validation or schema processing, without explicit operator action." Implementations **MAY** offer an opt-in mode that fetches non-local `$ref`s, but it MUST be disabled by default and SHOULD enforce an allowlist of hosts (or at minimum reject loopback, link-local, and private network addresses), apply timeouts and size limits, and log dereferenced URIs.
- Schemas that fail to validate due to an unresolved external `$ref` SHOULD be rejected rather than silently treated as permissive.

Composition-Keyword Resource Use

Composition keywords (`anyOf`, `oneOf`, `allOf`, `if` / `then` / `else`) and `$defs` enable expressive schemas, but pathological combinations can be expensive to validate. Implementations SHOULD apply reasonable bounds — for example, a maximum schema depth, a cap on the total number of subschemas, or a per-validation time budget — to prevent a malicious tool definition from acting as a CPU DoS vector against the validator.

Reference Implementation

TypeScript SDK

A reference implementation demonstrating the loosened type restrictions:

- **Branch**: olaservo/typescript-sdk@sep-834-v1x
- **npm**: `@olaservo/mcp-sdk@1.25.2-sep834.4`
- **Key changes**:
  - `inputSchema`: Retains `type: "object"` but allows any additional JSON Schema properties (compositions like `oneOf` / `anyOf`)
  - `outputSchema`: Any valid JSON Schema object (arrays, primitives, objects, compositions)
  - `structuredContent`: Any JSON value (objects, arrays, or primitives)
  - McpServer high-level API updated to support array and primitive outputSchema

Everything Server Demo Tools

Three demo tools added to the `everything` server demonstrating SEP-2106 capabilities:

- **Branch**: olaservo/servers@sep-834-json-schema-2020-12
- **npm**: `@olaservo/mcp-server-everything-sep834@1.1.0-sep834.1`
- **Tools**:
  - `get-weather-forecast`: Returns **raw array** of hourly forecasts directly in `structuredContent`
    - Matches the exact example from SEP-2106's Motivation section
    - `outputSchema`: `z.array(HourlyForecastSchema)` - array type at root
    - `structuredContent`: `[{hour, temp, conditions}, ...]` - direct array
  - `find-by-id-or-name`: Demonstrates flexible input patterns (accepts `id` OR `name`)
  - `get-count`: Returns **raw number** directly in `structuredContent` (not wrapped in object)
    - `outputSchema`: `z.number()` - primitive type at root
    - `structuredContent`: `42` - direct primitive

Related Links

- Original PR: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/881
- Related issue: https://github.com/modelcontextprotocol/modelcontextprotocol/issues/834
- `outputSchema` type restriction inconsistency: https://github.com/modelcontextprotocol/modelcontextprotocol/issues/1906
- TypeScript SDK schema types: https://github.com/modelcontextprotocol/typescript-sdk/issues/1149
- SEP-2200 (Clarify Tool Result Content and Model Visibility): https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2200

Implementation Guidance

SDK implementations will need to:

1. Update `inputSchema` types to retain `type: "object"` but allow any additional JSON Schema properties
2. Update `outputSchema` types to allow any valid JSON Schema (remove `type: "object"` constraint)
3. Update `structuredContent` types to accept any valid JSON value
4. Update JSON Schema definitions accordingly

Acknowledgments

This proposal builds on discussions in GitHub issue #834 and incorporates feedback from the MCP community.

SEP-2133 Extensions

Final · Standards Track · Created 2025-01-21

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-01-21
- **Author(s)**: Peter Alexander (@pja-ant)
- **Sponsor**: None (seeking sponsor)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2133

Abstract

This SEP establishes a lightweight framework for extending the Model Context Protocol through optional, composable extensions. This proposal defines a governance model and presentation structure for extensions that allows the MCP ecosystem to evolve while maintaining core protocol stability. Extensions enable experimentation with new capabilities without forcing adoption across all implementations, providing clear extension points for the community to propose, review, and adopt enhanced functionality.

This SEP defines both official extensions (maintained by MCP maintainers) and experimental extensions (an incubation pathway for Working Groups and Interest Groups to prototype and collaborate on extension ideas before formal acceptance). Externally maintained extensions will likely come at a later stage.

Motivation

MCP currently lacks any form of guidance on how extensions are to be proposed or adopted. Without a process, it is unclear how these extensions are governed, what expectations there are around implementation, how they should be referenced in the specification, etc.

Specification

Definition

An MCP extension is an optional addition to the specification that defines capabilities beyond the core protocol. Extensions enable functionality that may be modular (e.g., distinct features like authentication), specialized (e.g., industry-specific logic), or experimental (e.g., features being incubated for potential core inclusion).

Extensions are identified using a unique *extension identifier* with the format: `{vendor-prefix}/{extension-name}`, e.g. `io.modelcontextprotocol/oauth-client-credentials` or `com.example/websocket-transport`. The names follow the same rules as the _meta keys, except that the prefix is mandatory.

To prevent identifier collisions, the vendor prefix SHOULD be a reversed domain name that the extension author owns or controls (similar to Java package naming conventions). For example, a company owning `example.com` would use `com.example/` as their prefix.

Breaking changes MUST use a new identifier, e.g. `io.modelcontextprotocol/oauth-client-credentials-v2`. A breaking change is any modification that would cause existing compliant implementations to fail or behave

incorrectly, including: removing or renaming fields, changing field types, altering the semantics of existing behavior, or adding new required fields.

Extensions may have settings that are sent in client/server messages for fine-grained configuration.

This SEP defines *Official Extensions* and *Experimental Extensions*. Experimental extensions are maintained within the MCP organization as an incubation pathway but are not yet officially accepted. *Unofficial extensions* are not recognized by MCP governance and may be introduced and governed by developers outside the MCP organization.

Official Extensions

Official extensions live inside the MCP github org at https://github.com/modelcontextprotocol/ and are officially developed and recommended by MCP maintainers. Official extensions use the `io.modelcontextprotocol` vendor prefix in their extension identifiers.

An *extension repository* is a repository within the official modelcontextprotocol github org with the `ext-` prefix, e.g. https://github.com/modelcontextprotocol/ext-auth.

- Extension repositories are created at the core maintainers discretion with the purpose of grouping extensions in a specific area (e.g. auth, transport, financial services).
- A repository has a set of maintainers (identified by MAINTAINERS.md) appointed by the core maintainers that are responsible for the repository and extensions within it (e.g. ext-auth MAINTAINERS.md, ext-apps MAINTAINERS.md).
- Extensions SHOULD have an associated working group or interest group to guide their development and gather community input.

An *extension* is a versioned specification document within an extension repository, e.g. https://github.com/modelcontextprotocol/ext-auth/blob/main/specification/draft/oauth-client-credentials.mdx

- Extension specifications MUST use the same language as the core specification (i.e. [BCP 14] [RFC2119] [RFC8174]) and SHOULD be worded as if they were part of the core specification.

While day-to-day governance is delegated to extension repository maintainers, the core maintainers retain ultimate authority over official extensions, including the ability to modify, deprecate, or remove any extension.

Experimental Extensions

Experimental extensions provide an incubation pathway for Working Groups (WGs) and Interest Groups (IGs) to facilitate discovery, prototype ideas, and collaborate on extension concepts before formal SEP submission. Experimental extensions allow cross-company collaboration under neutral governance with clear anti-trust protection and IP clarity.

An *experimental extension repository* is a repository within the official modelcontextprotocol github org with the `experimental-ext-` prefix, e.g. `https://github.com/modelcontextprotocol/experimental-ext-interceptors`.

- Any maintainer MAY create an experimental extension repository while the associated SEP is still in draft state (or before a SEP has been submitted).
- Experimental extensions MUST be associated with a Working Group or Interest Group, whose maintainers are responsible for day-to-day governance of the repository.
- Experimental extension repositories MUST clearly indicate their experimental/non-official status (e.g., in the README) to avoid confusion with official extensions.
- Any published packages from experimental extensions MUST use naming that clearly indicates their experimental status.
- Core maintainers retain oversight of experimental extension repositories, including the ability to archive or remove them.

To graduate an experimental extension to official status, the standard SEP process (Extensions Track) applies. The experimental repository and any reference implementations developed during incubation MAY be referenced in the SEP to demonstrate the extension's practicality.

Lifecycle

Creation

Extensions MAY optionally begin as experimental extensions (see *Experimental Extensions* section) to facilitate prototyping and collaboration before formal submission. This incubation period is encouraged but not required.

To become an official extension, extensions are created via a SEP in the main MCP repository using the standard SEP guidelines but with a new type: **Extensions Track**. This type follows the same review and acceptance process as Standards Track SEPs, but clearly indicates that the proposal is for an extension rather than a core protocol addition. The SEP must identify the Working Group and Extension Maintainers that will be responsible for the extension. See SEP-2148 for how maintainers are appointed.

Extension SEPs:

- SHOULD be discussed and iterated on in a relevant working group prior to submission.
- MUST have at least one reference implementation in an official SDK prior to review to ensure the extension is practical and implementable.
- MAY reference an existing experimental extension repository and implementations developed during incubation.
- Will be reviewed by the Core Maintainers, who have the final authority over its inclusion as an Official Extension.

Once approved, the author SHOULD produce a PR that introduces the extension to the extension repository and reference in the main spec (see *Spec Recommendation* section). Approved extensions MAY be implemented in additional clients / servers / SDKs (see *SDK Implementation*).

Iteration

Once accepted, extensions may be iterated on without further review from the Core Maintainers. The extension repository maintainers are responsible for the review and acceptance of changes to an extension and SHOULD coordinate change via the relevant working group(s). As extensions are independent of the core protocol, extensions may be updated and deployed at any time, but changes MUST ensure they account for backwards compatibility in their design.

Promotion to Core Protocol (Optional)

Eventually, some extensions MAY transition to being core protocol features. This SHOULD be treated as a Standards Track SEP with separate core maintainer review. Note that not all extensions are suitable for inclusion in the core protocol (e.g. those specific to an industry) and may remain as extensions indefinitely.

Spec Recommendation

Extensions will be referenced from a new page on the MCP website at openmodelcontextprotocol.org/extensions (to be created) with links to their specification.

Links to relevant extensions MAY also be added to the core specification as appropriate (e.g. https://openmodelcontextprotocol.org/specification/draft/basic/authorization may link to ext-auth extensions), but they MUST be clearly advertised as optional extensions and SHOULD be links only (not copies of specification text).

SDK Implementation

SDKs MAY implement extensions. Where implemented, extensions MUST be disabled by default and require explicit opt-in. SDK documentation SHOULD list supported extensions.

SDK maintainers have full autonomy over extension support in their SDKs:

- Maintainers are solely responsible for the implementation and maintenance of any extensions they choose to support.
- Maintainers are under no obligation to implement any extension or accept contributed implementations. Extension support is not required for 100% protocol conformance or the upcoming SDK conformance tiers.
- This SEP does not prescribe how SDKs should structure or package extensions. Maintainers may provide extension points, plugin systems, or any other mechanism they see fit.

Evolution

All extensions evolve **independently** of the core protocol, i.e. a new version of an extension MAY be published without review by the core maintainers. Minor updates, bug fixes, and non-breaking enhancements to an extension do not require a new SEP; these changes are managed by the extension repository maintainers.

Extensions SHOULD be versioned, but exact versioning approach is not specified here.

Negotiation

Clients and servers advertise their support for extensions in the ClientCapabilities and ServerCapabilities fields respectively, and in the Server Card (currently in progress).

A new "extensions" field will be introduced to each that is a map of *extension identifiers* to per-extension settings objects. Each extension specifies the schema of its settings object; an empty object indicates no settings.

Client Capabilities

Clients advertise extension support in the `initialize` request:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "roots": {
        "listChanged": true
      },
      "extensions": {
        "io.modelcontextprotocol/ui": {
          "mimeTypes": ["text/html;profile=mcp-app"]
        }
      }
    },
    "clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    }
  }
}
```

Server Capabilities

Servers advertise extension support in the `initialize` response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "protocolVersion": "2025-06-18",
    "capabilities": {
      "tools": {},
      "extensions": {
        "io.modelcontextprotocol/ui": {}
      }
    },
    "serverInfo": {
      "name": "ExampleServer",
      "version": "1.0.0"
    }
  }
}
```

Server-Side Capability Checking

Servers SHOULD check client capabilities before offering extension-specific features:

```
const hasUISupport = clientCapabilities?.extensions?.[
  "io.modelcontextprotocol/ui"
]?.mimeTypes?.includes("text/html;profile=mcp-app");

if (hasUISupport) {
  // Register tools with UI features
} else {
  // Register text-only fallback
}
```

Graceful Degradation

If one party supports an extension but the other does not, the supporting party MUST either revert to core protocol behavior or reject the request with an appropriate error if the extension is mandatory. Extensions SHOULD document their expected fallback behavior. For example, a server offering UI-enhanced tools should still return meaningful text content for clients that do not support the UI extension, while a server requiring a specific authentication extension MAY reject connections from clients that do not support it.

Legal Requirements

Trademark Policy

- Use of MCP trademarks in extension identifiers does not grant trademark rights. Third parties may not use 'MCP', 'Model Context Protocol', or confusingly similar marks in ways that imply endorsement or affiliation.
- MCP makes no judgment about trademark validity of terms used in extensions.

Antitrust

- Extension developers acknowledge that they may compete with other participants, have no obligation to implement any extension, are free to develop competing extensions and protocols, and may license their technology to third parties including for competing solutions.
- Status as an official extension does not create an exclusive relationship.
- Extension repository maintainers act in individual capacity using best technical judgment.

Licensing

Official extensions MUST be available under the Apache 2.0 license.

Contributor License Grant

By submitting a contribution to an official MCP extension repository, you represent that:

1. You have the legal authority to grant the rights in this agreement
2. Your contribution is your original work, or you have sufficient rights to submit it
3. You grant to Linux Foundation and recipients of the specification a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable license to:
    - Reproduce, prepare derivative works of, publicly display, publicly perform, sublicense, and distribute the contribution
    - Make, have made, use, offer to sell, sell, import, and otherwise transfer implementations

No Other Rights

Except as explicitly set forth in this section, no other patent, trademark, copyright, or other intellectual property rights are granted under this agreement, including by implication, waiver, or estoppel.

Not Specified

This SEP does not specify all aspects of an extension system. The following is an incomplete list of what this SEP does not address:

- **Schema**: we do not specify a mechanism for extensions to advertise how they modify the schema.
- **Dependencies**: we do not specify if/how extensions may have dependencies on specific core protocol versions, or interdependencies with other extensions (or versions of extensions).
- **Profiles**: we do not specify a way of grouping extensions.

These are omitted not because they are unimportant, but because they may be added later and the goal of this SEP is simply to get some initial extension structure off the ground and defers detailed technical discussion around more complex/debatable aspects of extensions.

Rationale

This design for extensions uses the following principles:

- **Start simple**: the intention is to have a relatively simple mechanism that allows people to start building and proposing extensions in a structured way.
- **Clear governance**: For now, the focus is on clear governance and less on implementation details.
- **Refine later**: Over time, once we have more experience with extensions, we can adjust the approach appropriately.

Some specific design choices:

- **Why extension repositories instead of individual/independent extensions?** Repositories provide a natural group and governance structure that allows for the repository maintainers to enforce structure and conformity to extensions. It avoids a failure case of different extensions in an area working in incompatible ways. Also provides a way to delegate much of the governance work.
- **Why not require core maintainer review for official extensions?** Delegated reviews allows for extensions to evolve autonomously without being bottlenecked on core maintainer review, which is already a (often months) long process.
- **Why separate versioning?** Extensions are additions to the spec and optional so there is no need to tie versions together. Separate versions allow for more rapid iteration.

Backward Compatibility

The extension framework itself is purely additive to the core protocol, so there are no backwards compatibility concerns with the core specification.

The design described in this SEP is consistent with existing official extensions (ext-apps and ext-auth), which already use the patterns specified here for capability negotiation and extension identifiers.

However, individual extensions may have their own backwards compatibility concerns. Extensions MUST consider and account for backwards compatibility in their design, both across core protocol versions and extension versions. Breaking changes within an extension MUST use a new extension identifier (see *Definition* section). Extensions SHOULD also document their approach to backwards compatibility and stability (e.g. an extension MAY advertise itself as "experimental" indicating that it may break without notice).

Security Implications

Extensions MUST implement all related security best practices in the area that they extend.

Clients and servers SHOULD treat any new fields or data introduced as part of an extension as untrusted and SHOULD comprehensively validate them.

Reference Implementation

To be provided.

SEP-2148 MCP Contributor Ladder

Final · Process · Created 2026-01-15

- **Status**: Final
- **Type**: Process
- **Created**: 2026-01-15
- **Author(s)**: David Soria Parra (@dsp-ant), Sarah Novotny (@sarahnovotny)
- **Sponsor**: David Soria Parra (@dsp-ant)
- **PR**: https://github.com/modelcontextprotocol/specification/pull/2148

Abstract

This SEP establishes a formal contributor ladder for the Model Context Protocol project, defining clear roles, responsibilities, and advancement criteria from first-time contributor through Core Maintainer. The ladder provides transparent pathways for community members to understand how they can grow their involvement and influence within the project.

This SEP is a companion to SEP-2149: MCP Group Governance and Charter Template, which defines how Working Groups and Interest Groups operate. The two SEPs intersect: WG/IG leadership requires Member status on this ladder, and group participation is a recognized pathway to ladder advancement.

Motivation

As MCP adoption grows, the project needs a clear framework for:

1. **Contributor Development**: Community members lack visibility into how to grow their involvement and influence within the MCP project. A defined ladder shows the path from first contribution to project leadership.
2. **Trust Building**: Merge rights and other high-privilege responsibilities are earned through demonstrated commitment and good judgment over time. A graduated system ensures contributors are set up for success and are trusted by existing maintainers and broader community before taking on greater ownership of the project.
3. **Organizational Diversity**: With multiple organizations contributing to MCP, the project needs mechanisms to prevent organizational capture while welcoming participation from outside Anthropic.
4. **Scalability**: Core Maintainer bandwidth is limited. Delegating authority to Maintainers and Working/Interest Group Leads through clear scope definitions enables the project to scale.
5. **Recognition**: Contributors invest significant effort in MCP. Formal recognition through defined roles acknowledges their contributions and encourages sustained engagement.

Without a contributor ladder, advancement decisions become ad-hoc, potentially inconsistent, and opaque to the community.

Specification

Guiding Principles

The contributor ladder operates under these principles:

- **Earned Trust**: Advancement based on demonstrated meaningful contributions that align with the project goals, good judgment, and sustained engagement, not tenure alone
- **Multiple Growth Pathways**: Code, specification work, documentation, and community building all lead to advancement
- **Transparency**: Criteria for advancement are explicit and consistently applied
- **Alignment With MCP Goals**: Individual contributors must demonstrate commitment to advance and evolve MCP project components beyond one's employer's interests

Role Definitions

Role	Summary	Key Privileges	Minimum Timeline
Contributor	Anyone who contributes to MCP	Submit issues, PRs, participate in discussions	Immediate
Member	Established, active contributor	GitHub org membership, triage rights, eligible for WG/IG leadership	2-3 months of meaningful contributions
Maintainer	Area steward with operational responsibility	Merge rights, release participation	6+ months as Member
Core Maintainer	Technical leadership and protocol stewardship	Final decision authority, governance participation	By invitation after sustained Maintainer contribution
Lead Maintainer	Ultimate project authority (founders)	All Core Maintainer privileges, veto authority, appoints Core Maintainers	Reserved for project founders — succession only
Community Moderator	CoC enforcement and community health	Moderation rights on community platforms, incident handling	Parallel track — Member status + appointment

Timelines listed are minimum contribution periods, not guarantees of advancement. They exist to protect the project from rapid privilege escalation and to ensure a high bar of demonstrated commitment. Actual advancement is discretionary and may take longer in practice; the only guarantee is that advancement will not happen on a shorter timescale than documented. Exceptions require explicit Core Maintainer approval with documented rationale.

Contributor

Anyone who has contributed to MCP in any form is a contributor. This includes:

- Opening issues or discussions

- Submitting pull requests
- Participating in working group discussions
- Improving documentation
- Helping other community members

No formal requirements, we welcome all contributions.

How to get started:

- Review the Contributing Guide
- Join community channels (Discord, GitHub Discussions)
- Look for issues tagged `good-first-issue` or `help-wanted`
- Attend working group meetings

Member

Members are established contributors who have demonstrated ongoing commitment to the success and growth of MCP.

Requirements:

- Multiple contributions to MCP (code, documentation, and/or community)
- At least one merged PR or accepted contribution
- Ongoing engagement with the MCP community and not just one-off contributions
- Enabled two-factor authentication on GitHub
- No objections from existing Members within 7 days

Sponsorship:

- Sponsored by two existing Members or Maintainers from different organizations
- or sponsored by one Core Maintainer or Lead Maintainer

Minimum timeline: 2-3 months of active participation

Responsibilities:

- Continue contributing in good faith
- Be responsive to assigned issues and PRs
- Follow community guidelines and code of conduct
- Help onboard new contributors when possible

Privileges:

- GitHub organization membership with triage rights
- Can be assigned to issues and PRs
- Can use shortcut approval or review commands on PRs, such as `/lgtm`
- Listed in community membership roster
- Can create PRs in restricted repositories
- Eligible for Working Group Lead or Interest Group Facilitator roles

Inactivity: Members with no contributions for 3 months may be moved to emeritus status. Re-engagement follows a simplified re-familiarization process.

Maintainer

Maintainers are trusted stewards who take operational responsibility for specific areas.

Requirements:

- Member for at least 6 months with sustained, high-quality contributions
- Demonstrated leadership in working groups or significant initiatives
- Shown ability to represent MCP's interests above an individual employer's or organization's interests
- Deep understanding of the MCP vision, roadmap, and design principles
- Understands how their area impacts real-world AI integration and model interaction patterns
- Completed security and governance onboarding

Sponsorship & Approval:

- Sponsored by an existing Maintainer or Core Maintainer
- Approved by Core Maintainers

Responsibilities:

- Operational ownership of area health (test stability, documentation currency)
- Responsible for the release processes and milestone planning of their respective scope
- Provide timely review of escalated decisions
- Active participation in governance discussions
- Mentor Members and develop future Maintainers
- Represent MCP in external contexts when appropriate
- Engage with the area ecosystem and stakeholders, understanding real-world usage, and representing community needs
- Ensure proposals reaching Core Maintainers are refined, well-considered, and account for ecosystem-wide impact
- Active participation in discussions on communication channels (GitHub issues, Discord)

Privileges:

- Merge privileges for owned areas
- Can sponsor new Maintainers
- Participate in roadmap and prioritization discussions
- Listed in `MAINTAINERS.md`
- Release participation

All pathways can lead to Maintainer, though the specific scope will align with the contribution type.

Inactivity: Maintainers with no contributions for 6 months may be moved to emeritus status following review by Core Maintainers. Merge rights are revoked upon emeritus transition. Re-engagement requires re-completing security and governance onboarding.

Core Maintainer

Core Maintainers hold final decision-making authority for the MCP technical direction. This is the highest level of trust in the community.

Note: The Core Maintainer role is intentionally limited to ensure coherent technical vision while the project scales. Core Maintainer bandwidth concerns are addressed through clearer delegation to Maintainers, Working Group Leads, and Interest Group Facilitators, not expansion of Core Maintainer numbers.

Requirements:

- Sustained contribution as Maintainer or similar roles over at least 6 months
- Demonstrated judgment on complex, project-wide decisions
- Trust and respect across organizational boundaries
- Deep commitment to MCP's long-term success

Appointment:

- Nominated by majority of Core Maintainers, approved by Lead Maintainers

- Or direct appointment by Lead Maintainers

When evaluating candidates, Core Maintainers should consider whether the current composition adequately represents the breadth of the MCP ecosystem, including enterprise adopters deploying MCP in production domains.

Responsibilities:

- Final technical decision authority for contested or cross-cutting issues
- Stewardship of project vision and design principles
- Governance and policy decisions
- External representation of MCP
- Succession planning and community health
- Ensure restraint and sustainability in protocol evolution
- Participation in Core Maintainer meetings and Core Maintainer meetups

Privileges:

- Final approval on breaking changes and major spec revisions
- Voting rights on SEPs (Specification Enhancement Proposals)
- Approval of Maintainers
- Governance voting rights / expectation of governance participation
- Administrative rights to all MCP GitHub repositories
- Listed in `MAINTAINERS.md` as Core Maintainer

Inactivity: Core Maintainers with no participation in governance or technical decisions for 6 months may be moved to emeritus status following review by Lead Maintainers. Given the trust and visibility of this role, Core Maintainers are expected to proactively communicate reduced availability.

Lead Maintainer

Lead Maintainers hold ultimate authority over MCP's direction and governance. This is a lifetime appointment reserved for project founders. There is no defined advancement path to this role; it is only assumed through succession when necessary (see Succession).

Responsibilities:

- All Core Maintainer responsibilities
- Appointment and removal of Core Maintainers
- Final authority on contested governance decisions
- Project-wide strategic direction

Privileges:

- Can act alone where Core Maintainers require multiple approvals
- Veto authority over any decision
- Appointment of successor

Succession

If a Lead Maintainer leaves their role for any reason, the succession process begins upon their written notice or, if unable to provide notice, upon a determination by the remaining Lead Maintainer(s) or Core Maintainers that the Lead Maintainer is unable to continue serving.

If one or more Lead Maintainer(s) remain, they shall appoint a successor (by majority vote if multiple), and the remaining Lead Maintainer(s) will continue to govern until a successor is appointed.

If no Lead Maintainers remain, the Core Maintainers shall appoint a successor by majority vote within 30 days, and the project operates by two-thirds vote of Core Maintainers until a new Lead Maintainer is appointed.

Advancement Process

Self-Nomination vs. Recognition

Contributors may either:

1. **Self-nominate** when they believe they meet the requirements
2. **Be nominated** by a sponsor who has observed their contributions

Both paths are equally valid. Self-nomination is encouraged and preferred, as it demonstrates initiative and self-awareness of the contribution scope.

Process Steps

1. **Nomination**: Nominee or sponsor opens an issue using the nomination template, including links to contributions demonstrating requirements and sponsor confirmations
2. **Community Review**: 7-day period for community input
3. **Decision**: Approving authority reviews and decides
4. **Onboarding**: New role-holder receives appropriate access and onboarding

Advancement To	Approved By
Member	2 existing Members+ from different organizations, **or** 1 Core/Lead Maintainer
Maintainer	1 Maintainer or Core Maintainer sponsor + Core Maintainer approval
Core Maintainer	Lead Maintainers
Community Moderator	1 Core Maintainer or Lead Maintainer

Self-nomination is encouraged, but nominees must still secure the required sponsorship. Sponsors confirm support in the nomination issue.

Decision-Making & Escalation

Delegation as Default

MCP operates on a principle of delegation: decisions should be made at the lowest appropriate level. This enables the project to move quickly while preserving Core Maintainer bandwidth for cross-cutting concerns.

- **Maintainers, WG Leads, and IG Facilitators** handle day-to-day decisions within scope
- **Core Maintainers** intervene on escalation, cross-cutting issues, or when required (spec changes, Maintainer approval)
- **Lead Maintainer** intervenes only on contested governance decisions or when Core Maintainers cannot reach consensus

When in doubt, make the decision at your level and document it. Escalate only when blocked, when the decision has project-wide implications, or when explicitly required by process.

The detailed escalation procedure for Working Group and Interest Group disputes — including the designation of a Core Maintainer without shared organizational affiliation to resolve the issue — is defined in SEP-2149 §1.5.

Escalation Matrix

Issue Type	First Escalation	Second Escalation	Timeline
Technical disagreement in PR	Maintainer in scope	Core Maintainer	5 business days
Technical disagreement in WG	WG Lead	Core Maintainer	5 business days
Technical disagreement in IG	IG Facilitator	Core Maintainer	5 business days
Disagreement with WG Lead / IG Facilitator	Core Maintainer	Lead Maintainer	7 business days
Disagreement with Maintainer decision	Core Maintainer	Lead Maintainer	7 business days
Core Maintainer disagreement	Lead Maintainer	N/A	10 business days
Code of Conduct violation	Community Moderator	Core Maintainer	Immediate
Security issue	Core Maintainer	Lead Maintainer	Immediate

Escalation process:

1. Document the decision, options considered, and points of disagreement
2. Present to the escalation authority with a clear ask
3. Escalation authority either: (a) provides binding guidance, (b) requests more information, or (c) escalates further if needed

Contribution Pathways

MCP values diverse contributions. Here are recognized pathways to advancement:

Code Contributions

- SDK development (TypeScript, Python, etc.)
- Testing infrastructure
- Tooling and developer experience

Specification Work

- Drafting or refining spec text
- SEP authorship or co-authorship
- Protocol design participation

- Compatibility analysis

Documentation

- User guides and tutorials
- API documentation
- Architecture documentation
- Maintaining content currency

Community Building

- Onboarding new contributors
- Working group facilitation
- Community support (Discord, GitHub discussions)
- Event organization or representation

Quality & Security

- Bug triage and reproduction
- Security review and analysis
- Test coverage improvement
- Release validation

Working Group and Interest Group Leadership

Working Group (WG) Leads and Interest Group (IG) Facilitators are a special form of community leadership that doesn't require Maintainer status. WG/IG leadership focuses on facilitation and coordination rather than merge authority. The full governance rules for WGs and IGs — including participation tiers, decision-making process, meeting requirements, and lifecycle — are defined in SEP-2149: MCP Group Governance and Charter Template.

Requirements:

- Member status minimum
- Demonstrated sustained engagement with the WG/IG's scope
- Good facilitation and communication skills
- Ability to represent multiple perspectives fairly
- Group and its leadership are sponsored by at least two Core Maintainers or one Lead Maintainer

Relationship to Contributor Ladder:

- WG Lead and IG Facilitator experience is valuable for advancement to Maintainer
- WG Leads and IG Facilitators without Maintainer status work with Maintainers for merge decisions
- WG Leads and IG Facilitators have authority over group operations but not spec approval
- WG Leads and Maintainers may sponsor SEPs
- WG Leads may triage SEPs in their scope area, including closing SEPs that do not fit the WG's roadmap (with documented rationale; authors may appeal to Core Maintainers)

Community Moderators

Community Moderators are trusted individuals who help keep the MCP community healthy, safe, and welcoming. This is a dedicated community role focused on moderation and Code of Conduct enforcement rather than technical contribution.

Requirements:

- Member status minimum
- Demonstrated good judgment and composure in community interactions
- Understanding of the MCP Code of Conduct and community guidelines
- Ability to handle sensitive situations with discretion and fairness

Sponsorship:

- Sponsored by a Core Maintainer or Lead Maintainer

Responsibilities:

- Monitor community channels (Discord, GitHub Discussions, etc.) for adherence to the Code of Conduct
- Handle Code of Conduct incident reports, including initial triage and response
- Escalate serious or complex incidents to Core Maintainers
- Help maintain a welcoming and inclusive environment for all community members
- Coordinate with other moderators to ensure consistent enforcement
- Document moderation actions and maintain confidentiality of incident details
- Recuse from any incident in which they are personally involved; such incidents are handled directly by Core Maintainers

Privileges:

- Moderation rights on community platforms (Discord, GitHub Discussions)
- Access to moderation tools and private moderation channels
- Authority to issue warnings, mute, or temporarily ban users for Code of Conduct violations
- Listed in community moderator roster

Relationship to Contributor Ladder:

- Community Moderator is a parallel track, not a prerequisite for technical advancement
- Moderator experience is valued for advancement to any role, particularly where community judgment is important
- Moderators may simultaneously hold other roles (Member, Maintainer, etc.)

Removal: Community Moderators may be removed by Core Maintainers for failure to uphold moderation standards or Code of Conduct violations. Moderators may step down voluntarily at any time.

Recognition and Visibility

The community recognizes contributors through:

- **Contributor lists** such as `MAINTAINERS.md`
- **GitHub teams** for appropriate access
- **Public acknowledgment** in release notes
- **Speaking opportunities** at community events
- **Badges** (if implemented) on community platforms

Stepping Down and Emeritus Status

Contributors may step down from roles for any reason. This is normal and healthy.

Process:

1. Notify relevant leadership (WG Lead, IG Facilitator, Maintainer, or Core Maintainer as appropriate)
2. Help transition any ongoing work
3. Move to emeritus status

Emeritus:

- Recognized for past contributions
- May return to active status with abbreviated re-onboarding
- No ongoing responsibilities or privileges

Involuntary Removal: In cases of code of conduct violations or sustained non-participation, roles may be revoked following appropriate review processes.

Rationale

Why a Formal Ladder?

Informal advancement creates inconsistency and opacity. A formal ladder:

- Sets clear expectations for all parties
- Provides a common vocabulary for discussing advancement
- Creates accountability in advancement decisions
- Enables self-nomination, reducing gatekeeping

Why Minimum Timelines?

Timelines are floors, not targets. They exist for security and trust-building:

- Trust is built through demonstrated behavior over time
- Security risks increase with rapid privilege escalation
- Deep project understanding requires sustained engagement
- Behavior patterns only become visible over longer periods

Meeting a minimum timeline does not create an entitlement to advancement; it establishes eligibility for consideration. Exceptions to minimums require explicit Core Maintainer approval with documented rationale.

Why Two-Organization Sponsorship?

Requiring sponsors from different organizations:

- Prevents organizational capture of the contributor base
- Ensures contributors are recognized beyond their employer
- Maintains diverse perspectives in advancement decisions

Model Inspiration

This ladder is modeled on Kubernetes community membership structures and adapted for MCP's needs and stage of development.

Backward Compatibility

This SEP establishes new processes without modifying existing structures. Current contributors retain their existing access and standing.

Security Implications

This SEP directly addresses security through:

- Graduated privilege escalation with timeline requirements
- Two-factor authentication requirement for Members
- Multi-organization sponsorship to prevent capture
- Security onboarding requirement for Maintainers

Reference Implementation

Upon acceptance, this SEP will be implemented by:

1. Adding the contributor ladder to `docs/community/contributor-ladder.mdx`
2. Creating nomination issue templates in `.github/ISSUE_TEMPLATE/` (see Appendix for checklist templates)
3. Updating `MAINTAINERS.md` format to reflect role distinctions

Appendix: Checklist Templates

Member Nomination Checklist

```
**Nominee:** [GitHub handle]
**Sponsors:** [GitHub handles]
  - **Organizations represented:** [Must be 2+ different orgs among sponsors]

**Contributions:**
- [ ] Link to merged PR(s)
- [ ] Link to issues filed/triaged
- [ ] Link to discussions participated in
- [ ] Duration of participation: [X months]

**Sponsor Attestations:**
Sponsors confirm
- [ ] Sponsors confirm nominee demonstrates community values
- [ ] Sponsors confirm nominee demonstrates sustained engagement
```

Maintainer Nomination Checklist

```
**Nominee:** [GitHub handle]
**Scope:** [Specific area]
**Sponsor:** [GitHub handle, must be Maintainer or Core Maintainer]

**Requirements:**
- [ ] Member for 6+ months with sustained, high-quality contributions
- [ ] Links to demonstrated leadership in WG, IG, or significant initiatives
- [ ] Evidence of representing MCP's interests above employer/organization interests
- [ ] Deep understanding of MCP vision, roadmap, and design principles
- [ ] Security and governance onboarding completed (or scheduled)

**Core Maintainer Approval:**
- [ ] Approved by Core Maintainers
```

Community Moderator Nomination Checklist

```
**Nominee:** [GitHub handle]
**Sponsor:** [GitHub handle, must be Core Maintainer or Lead Maintainer]

**Requirements:**
- [ ] Member status
- [ ] Links to demonstrated good judgment and composure in community interactions
- [ ] Confirmed understanding of the MCP Code of Conduct and community guidelines

**Sponsor Attestation:**
- [ ] Sponsor confirms nominee can handle sensitive situations with discretion and
fairness
```

SEP-2149 MCP Group Governance and Charter Template

Final · Process · Created 2025-01-15

- **Status**: Final
- **Type**: Process
- **Created**: 2025-01-15
- **Author(s)**: David Soria Parra (@dsp-ant), Sarah Novotny (@sarahnovotny)
- **Sponsor**: David Soria Parra (@dsp-ant)
- **PR**: https://github.com/modelcontextprotocol/specification/pull/2149

Abstract

This SEP establishes governance rules and a standardized charter template for MCP's two collaborative group types: **Working Groups (WGs)** and **Interest Groups (IGs)**. Working Groups produce concrete deliverables — SEPs, implementations, and code. Interest Groups facilitate discussion and knowledge-sharing to identify problems and gather requirements. The governance rules define the requirements that all groups must follow, with lighter expectations for IGs where appropriate. The charter template defines the structure each group uses to document its specific mission, scope, leadership, and work. Together they address community feedback about unclear authority delegation and inconsistent processes across groups.

This SEP is a companion to SEP-2148: MCP Contributor Ladder, which defines the org-wide contributor roles (Member, Maintainer, Core Maintainer, Lead Maintainer) referenced throughout this document. Group leadership roles intersect with the contributor ladder: WG Leads and IG Facilitators must hold at least Member status on the ladder, and group participation is a recognized pathway to ladder advancement.

Motivation

Community interviews and feedback identified several challenges with the current group structure:

1. **Unclear Authority**: It's not always clear what decisions a working group can make autonomously versus what requires Core Maintainer approval. This leads to hesitation and bottlenecks.
2. **Inconsistent Decision-Making**: Different groups operate with different norms. Decisions made in one meeting may be contradicted in another, with no clear process for resolution.
3. **Participation Confusion**: Community members are uncertain about who should participate in groups, what levels of involvement exist, and how to become more involved.
4. **Scope Creep**: Without explicit boundaries, groups may gradually expand into areas owned by other groups or outside their mandate.
5. **Missing Escalation Paths**: When groups get stuck, there's no clear path to resolution, leading to prolonged disagreements or abandoned initiatives.
6. **WG/IG Distinction**: The difference between Working Groups and Interest Groups is not always clear to participants, leading to mismatched expectations about outputs and commitment.

A standardized charter template and shared governance rules address these issues by establishing consistent processes across all groups while requiring each group to explicitly define its specific scope and boundaries.

Specification

MCP maintains two types of collaborative groups:

- **Working Groups (WGs)** produce concrete deliverables — SEPs, reference implementations, and code. Active contribution is expected.
- **Interest Groups (IGs)** facilitate discussion and knowledge-sharing around a topic area. They produce problem statements, use cases, and recommendations. Active contribution is expected.

This specification has two parts:

1. **Group Governance** — rules that apply to all MCP groups (WGs and IGs), with differences noted where applicable
2. **Charter Template** — the structure each group fills in to define its specific mission, scope, and operations

---

Part 1: Group Governance

The following rules apply to all MCP Working Groups and Interest Groups. Individual charters cannot override these requirements. Where rules differ between WGs and IGs, this is noted explicitly.

1.1 Leadership

Each group has one or more **Leads** (referred to as **Facilitators** for Interest Groups).

Requirements for all Leads and Facilitators:

- Hold at least Member status on the MCP Contributor Ladder
- Demonstrated sustained engagement with the group's scope area
- Ability to facilitate across organizational boundaries
- Commitment to running the group's operations
- Group and its leadership sponsored by at least two Core Maintainers or one Lead Maintainer

Additional requirements for WG Leads:

- Commitment to 2-3 hours/week for WG activities

1.2 Leadership Responsibilities

All Leads are responsible for:

- Schedule and facilitate regular meetings
- Set agendas in collaboration with participants and publish them in advance
- Ensure meeting notes are published within 48 hours
- Maintain the group's documentation
- Maintain a members list and respective access list in https://github.com/modelcontextprotocol/access
- Proactively recruit and retain broad, representative membership across organizations and perspectives

WG Leads are additionally responsible for:

- Drive proposals through the SEP (Specification Enhancement Proposal) process to resolution
- Triage SEPs in the WG's scope area, including closing SEPs that do not fit the roadmap (with documented rationale; authors may appeal to Core Maintainers)

- Escalate blocked decisions to Core Maintainers with clear context
- Maintain the working group's roadmap
- Solicit feedback from one or more Core Maintainers on the general direction of the group on a continuous basis
- Provide quarterly status updates to the Community and Core Maintainer Group

1.3 Participation Levels

All groups use the following participation tiers. Note that **WG Member** is a group-specific participation level distinct from the org-wide **Member** role defined in the Contributor Ladder — an individual may be a WG Member in a specific group without holding org-wide Member status, and vice versa.

Level	Description	Privileges
Observer	Anyone interested in following the group's work	Read access, may attend meetings, limited discussion participation
Participant	Active contributor to group discussions	Can propose agenda items, participate in async votes
WG Member	Sustained contributor with demonstrated expertise	Counted for quorum (WGs only)
Lead/Facilitator	Operational leadership of the group	Sets agenda, facilitates, escalates

Interest Groups primarily operate with Observers, Participants, and Facilitators. IGs may adopt the WG Member tier if their work warrants formal decision-making, but are not required to.

Becoming a WG Member (WGs, and IGs that adopt the WG Member tier):

- Sustained participation over 3 months
- Meaningful contributions (code, spec text, reviews, or documentation)
- Nomination by existing WG Member or Lead
- No objections from Leads, Core Maintainers, or Lead Maintainers within 7 days

WG Member Responsibilities:

- Continue contributing in good faith
- Maintain name, organization, and Discord name in the respective group's member list

Active vs. Emeritus: WG Members who do not participate for 3 consecutive months are moved to emeritus status and may return by demonstrating renewed participation.

1.4 Decision-Making Process

This section applies primarily to Working Groups, which make binding decisions (consensus on technical designs, spec changes, etc.). Interest Groups typically operate by rough consensus in discussions and do not make binding decisions — their output is recommendations, problem statements, and use cases. IGs that adopt the WG Member tier may use this process for internal decisions.

WG Consensus is achieved through the following progression. Each step is attempted before moving to the next:

Step 1: Lazy Consensus (default)

- Proposals announced with clear deadline (5 days minimum for minor items, 10 days for significant items)

- Silence is consent
- Any WG Member may block with documented objection
- Blocks must propose alternatives or clear criteria for resolution
- If no blocks are raised by the deadline, the proposal is accepted

Step 2: Formal Vote (when lazy consensus is blocked)

A formal vote is triggered when:

- A WG Member blocks during the lazy consensus period
- A Lead or three or more WG Members request a formal vote

Voting rules:

- Quorum: 50% of active WG Members
- Passage: Simple majority for routine matters; 2/3 majority for scope changes
- Core Maintainer feedback is advisory unless explicitly stated as binding
- All votes documented with rationale

Step 3: Escalation (when voting does not resolve)

If a vote fails to resolve the matter (no quorum, does not pass, or the result is contested), the Lead escalates to Core Maintainers following the escalation path defined below.

1.5 Escalation Path

For technical and design disagreements within a group's scope, groups should resolve disagreements locally before involving Core Maintainers. For WGs, this means using the decision-making progression (lazy consensus → vote → escalation). For IGs, the Facilitator should attempt to find rough consensus before escalating.

Some disagreements are not appropriate for group-level resolution and should be escalated directly to Core Maintainers:

- Scope disputes (whether a topic falls within the group's charter)
- Authority disputes (whether the group has the right to decide a matter)
- Cross-group conflicts (disagreements spanning multiple WGs or IGs)
- Code of conduct or behavioral concerns
- Membership or participation disputes

When escalation is necessary:

1. Lead documents the decision, options considered, and points of disagreement
2. Lead presents the escalation to the Core Maintainer group with a clear ask
3. The Core Maintainer group designates a CM—who should not share organizational affiliation with the parties involved—to resolve the issue and report back to the group
4. The designated CM either: (a) provides binding guidance, (b) requests more information, or (c) recommends the full Core Maintainer group deliberate
5. Timeline: Escalations should receive initial response within 5 business days

1.6 Meeting Requirements

Leads determine meeting frequency, format, and duration based on the group's current needs and lifecycle stage. There is no fixed cadence requirement — a WG near a specification release may meet weekly, while an IG in early exploration may meet monthly or work primarily asynchronously.

Regardless of format or frequency, all group meetings must:

- Be open to all community participants (no closed or organization-internal meetings)
- Be published on meet.openmodelcontextprotocol.org at least 7 days in advance

- Have agendas published and publicly available. The agenda or a link to the agenda should be published as a GitHub Discussion in the Meeting Notes category
- Have notes published within 48 hours to the same discussion

Leads should actively involve WG Members and Participants in operational duties such as preparing agendas, taking meeting notes, and facilitating discussions.

1.7 Communication Channels

All groups use the following channels:

Channel	Purpose	Response Expectation
Discord `#{name}-wg` or `#{name}-ig`	Quick questions, coordination	Best effort
GitHub Discussions	Long-form technical discussion	Weekly triage

In addition to Discord, groups can establish a discussion category in the GitHub Discussions. Leads will be granted the appropriate roles to manage and moderate discussions.

1.8 Reporting

Working Groups provide quarterly updates (end of January, April, July, October) including:

- Progress against deliverables
- Blocked items and escalations
- Membership changes
- Upcoming priorities
- Resource needs

The quarterly updates are provided as a document posted in the GitHub Discussions category of the Working Group. They are optionally discussed with the Core Maintainers in a core maintainer meeting.

Interest Groups do not have formal reporting requirements but should keep their charter and member list current.

1.9 Lifecycle

Working Group Formation:

- There must be a widely acknowledged concern requiring coordination
- PR for creation of WG into `docs/community/<name>/overview.mdx`, gated by CODEOWNERS requiring approval by Maintainers
- PR for charter into `docs/community/<name>/charter.mdx`, gated by CODEOWNERS requiring approval from a single Core Maintainer (who should notify all Core Maintainers)
- Initial member list approved by WG Lead

Interest Group Formation:

- Fill out the creation template in the `#wg-ig-group-creation` channel on Discord
- A Core Maintainer reviews the proposal; the IG and its Facilitator(s) must be sponsored by at least two Core Maintainers or one Lead Maintainer
- Once sponsored, the Facilitator(s) organize the IG and create a charter

Retirement:

- **WGs**: WG Lead or Core Maintainer proposes retirement with rationale; Core Maintainer or Lead Maintainer approval required. WGs are also retired when they have no active work for a sustained period or have completed all planned deliverables.
- **IGs**: Core Maintainers or Lead Maintainers may retire an IG that is no longer active or needed.
- In both cases, documentation is archived and channels are marked inactive.

1.10 Charter Amendments

Changes to a group's charter (WG or IG) require:

- Proposal by Lead/Facilitator or Core Maintainer
- Approval by Core Maintainers

Part 2: Charter Template

Every MCP Working Group and Interest Group must maintain a charter document following this template structure. Charters are stored as MDX files at `docs/community/<group-name>/charter.mdx` in the modelcontextprotocol repository and added to the `docs/docs.json` file. A copyable version of this template was previously published in the community documentation.

The charter captures information specific to each group. Governance rules from Part 1 apply automatically and do not need to be repeated in the charter. Sections marked **(WG only)** are required for Working Groups but optional for Interest Groups.

1. Group Type

State whether this is a **Working Group** or an **Interest Group**.

2. Mission Statement

A 2-3 sentence summary of the group's purpose, articulating:

- The problem space being addressed
- Why cross-cutting collaboration is needed
- For WGs: what concrete deliverables the group will produce
- For IGs: what discussions and knowledge-sharing the group will facilitate

WG Example:

> The Transport Working Group exists to evolve MCP's transport mechanisms to support diverse deployment scenarios—from local subprocess communication to horizontally-scaled cloud deployments—while maintaining protocol coherence and backward compatibility.

IG Example:

> The Enterprise IG explores the challenges of deploying MCP in enterprise environments, gathering use cases and requirements to inform future specification work.

3. Scope

In Scope: Enumerated responsibilities.

For WGs, this includes:

- Specification Work: Specific spec sections or SEPs owned
- Reference Implementations: SDK components or reference implementations
- Cross-Cutting Concerns: Areas requiring coordination with other groups
- Documentation: Documentation responsibilities

For IGs, this includes:

- Topic areas for discussion
- Types of output (problem statements, use cases, recommendations)

Out of Scope: Explicit statements of what is NOT within the group's purview to prevent mission creep.

Related Groups: List of other WGs or IGs with intersecting work and nature of overlap.

4. Leadership

Leads/Facilitators table with:

- Role, Name, Organization, GitHub handle, Term

Leadership requirements and responsibilities are defined in the governance rules (Sections 1.1 and 1.2).

5. Authority & Decision Rights (WG only)

Each WG must explicitly define its decision authority. The decision-making process and escalation path are defined in the governance rules (Sections 1.4 and 1.5). This section documents which decisions the WG can make at which authority level.

Example:

Decision Type	Authority Level
Meeting logistics & scheduling	WG Leads (autonomous)
Proposal prioritization within WG	WG Leads (autonomous)
SEP triage & closure (in scope)	WG Leads (autonomous, with documented rationale)
Technical design within scope	WG consensus
Spec changes (additive)	WG consensus → Core Maintainer approval
Spec changes (breaking/fundamental)	WG consensus → Core Maintainer approval + wider review
Scope expansion	Core Maintainer approval required
WG Member approval	WG Member sponsors

IGs do not make binding decisions and do not need this section.

6. Membership

List current group members and their participation levels, if any. Leave out if no members exist yet. Participation tiers and membership criteria are defined in the governance rules (Section 1.3).

7. Operations

Document the group's current meeting approach. Meeting requirements and communication channels are defined in the governance rules (Sections 1.6 and 1.7).

Example:

Meeting	Frequency	Duration	Purpose
Working Session	Weekly/Biweekly	60 min	Technical discussion, proposal review
Office Hours	Monthly	30 min	Open Q&A for newcomers and observers

8. Deliverables & Success Metrics (WG only)

Active Work Items:

Item	Status	Target Date	Champion
SEP-XXX: Name	Draft/Review/Approved	Date	Name

Success Criteria: Measurable outcomes for WG success.

Quarterly reporting requirements are defined in the governance rules (Section 1.8).

IGs do not track formal deliverables but may list current discussion topics or planned outputs (problem statements, recommendations, etc.) in their charter.

9. Changelog

Track charter versions with date and changes.

Rationale

Why Separate Governance from Charter Template?

Separating fixed governance rules from the per-group charter template makes it clear what is consistent across all groups (decision-making, membership tiers, escalation) versus what each group defines for itself (scope, leadership roster, deliverables). This prevents groups from accidentally diverging on process while preserving flexibility where it matters.

Why Cover Both WGs and IGs?

Working Groups and Interest Groups serve different purposes but share common operational needs — leadership, meeting requirements, communication channels, and escalation paths. A unified governance framework ensures consistency while clearly articulating where IGs have lighter requirements (no formal decision authority, no deliverables tracking, no quarterly reporting).

Why a Standardized Template?

Standardization:

- Ensures all groups address critical governance questions
- Makes it easier for community members to understand any group's operations
- Reduces overhead for forming new groups
- Creates accountability through explicit documentation

Why Explicit Authority Tables?

The authority table directly addresses the "unclear authority" feedback. By enumerating decision types and required approvals, WGs and community members know exactly what can be decided autonomously versus what needs escalation. IGs are exempt from this because they produce recommendations, not binding decisions.

Why Tiered Participation?

Different engagement levels serve different community needs:

- **Observers** can learn without commitment
- **Participants** can contribute without full WG Member responsibilities
- **WG Members** take on accountability and get decision rights (primarily WGs)
- **Leads/Facilitators** provide operational continuity

Why Lazy Consensus as Default?

Lazy consensus:

- Enables efficient decision-making for routine matters
- Reduces meeting burden
- Documents decisions through announcement/deadline structure
- Preserves blocking rights for substantive concerns

Voting is reserved for contested or high-impact decisions.

Model Inspiration

This template is adapted from Kubernetes governance structures and tailored for MCP's specific needs identified through community interviews.

Backward Compatibility

Transition for Existing Groups

Working Groups and Interest Groups that exist at the time this SEP is accepted are grandfathered in — they are recognized as valid groups and do not need to re-apply through the formation process defined in Section 1.9.

However, existing groups must create a charter conforming to the template in Part 2 within **8 weeks** of this SEP's acceptance. During this transition period:

- Existing groups continue to operate under their current processes
- Leads/Facilitators are responsible for drafting the charter
- Core Maintainers will review and approve both WG and IG charters
- Groups that do not produce a charter within 8 weeks will be considered inactive and subject to retirement

Security Implications

No direct security implications. However, clear authority delegation and decision processes indirectly support security by ensuring decisions are made at appropriate levels with proper accountability.

Reference Implementation

This SEP is implemented by:

1. `docs/community/working-interest-groups.mdx` — governance rules (Part 1) published as the Working and Interest Groups page on openmodelcontextprotocol.org
2. `docs/community/charter-template.mdx` — copyable charter template (Part 2) linked from the above
3. Both pages added to `docs/docs.json` under the Community → Governance navigation group
4. Existing WGs and IGs must create conforming charters at `docs/community/<name>/charter.mdx` within 8 weeks of acceptance

SEP-2164 Standardize Resource Not Found Error Code

Final · Standards Track · Created 2026-01-28

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-01-28
- **Author(s)**: Peter Alexander (@pja-ant)
- **Sponsor**: None (seeking sponsor)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2164

Abstract

The current MCP specification recommends `-32002` as the error code for resource not found. However, `-32002` falls within the JSON-RPC "server error" range (`-32000` to `-32099`) which is reserved for implementation-defined errors, not protocol-level semantics. Additionally, SDK implementations are inconsistent — only 4 of 6 official SDKs use `-32002`, while the TypeScript SDK uses `-32602` and the Python SDK uses `0`.

This SEP standardizes on `-32602` (Invalid Params), the correct JSON-RPC error code for this case, and aligns the specification with the JSON-RPC standard.

Motivation

Current SDK implementations vary in their error handling for resource not found:

SDK	Current Error Code	Source
TypeScript	`-32602` (InvalidParams)	mcp.ts#L561
Python	`0` (generic)	server.py#L790
C#	`-32002` (custom RESOURCE_NOT_FOUND)	McpServerImpl.cs#L289
Rust	`-32002` (custom RESOURCE_NOT_FOUND)	model.rs#L450
Java	`-32002` (custom RESOURCE_NOT_FOUND)	McpAsyncServer.java#L732
Go	`-32002` (custom RESOURCE_NOT_FOUND)	server.go#L786
Kotlin	`-32603` (INTERNAL_ERROR)	Server.kt#L618-L621
PHP	`-32002` (custom RESOURCE_NOT_FOUND)	Error.php#L37
Ruby	N/A (left to implementor)	server.rb#L375-L379
Swift	N/A (no built-in handler)	N/A

This inconsistency means clients cannot reliably detect resource-not-found conditions across implementations. Of the 8 SDKs with built-in resource handling, four different error codes are used: `-32002` (C#, Rust, Java, Go, PHP), `-32602` (TypeScript), `-32603` (Kotlin), and `0` (Python). Ruby and Swift leave error handling to the server implementor. Clients that need to distinguish "resource not found" from other errors must handle all variants.

Specification

If the requested resource does not exist, servers MUST return a JSON-RPC error with code `-32602` (Invalid Params):

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "error": {
    "code": -32602,
    "message": "Resource not found",
    "data": {
      "uri": "file:///nonexistent.txt"
    }
  }
}
```

The `data` field SHOULD include the `uri` that was not found.

Servers MUST NOT return an empty `contents` array for a non-existent resource. An empty array is ambiguous — it could mean the resource exists but has no content, or that it doesn't exist at all.

Rationale

Why `-32602` (Invalid Params)?

`-32602` is the standard JSON-RPC error code for invalid parameters. A non-existent URI is semantically an invalid parameter — the client provided a URI that doesn't correspond to any resource. This aligns with the TypeScript SDK's existing behavior and avoids introducing custom error codes outside the JSON-RPC reserved range.

Why Not a Custom Error Code?

Several SDKs use `-32002` (RESOURCE_NOT_FOUND), but:

- Custom codes in the `-32000` to `-32099` range are "reserved for implementation-defined server errors" per JSON-RPC spec, not for protocol-level semantics
- Adding a protocol-defined custom code requires all clients to be updated to recognize it
- `-32602` already has the correct meaning and is universally understood by JSON-RPC libraries

Backward Compatibility

This changes what is specified — the current spec recommends `-32002`, and this SEP changes it to `-32602`. However, since the current recommendation is not consistently followed across SDKs (only 5 of 10 use `-32002`), clients cannot rely on any single error code today. This means the practical impact on clients is minimal — any

client robust enough to work across existing SDKs already handles multiple error codes or treats all errors generically.

Migration Path

1. SDKs should update their resource-not-found error code to `-32602`
2. During the transition, clients SHOULD handle both `-32602` and `-32002` as resource-not-found
3. The specification should document `-32602` as the canonical error code

Security Implications

None. This change only affects error code values, not access control or data exposure.

SEP-2207 OIDC-Flavored Refresh Token Guidance

Final · Standards Track · Created 2026-02-04

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-02-04
- **Author(s)**: Wils Dawson (@wdawson)
- **Sponsor**: Paul Carleton (@pcarleton)
- **PR**: #2207

Abstract

This proposal provides guidance for MCP implementations regarding refresh token issuance and requests, particularly when Authorization Servers support the `offline_access` scope. The `offline_access` scope originated in OIDC but can be adopted by any OAuth 2.1 Authorization Server as a mechanism to let clients explicitly request refresh tokens. This SEP clarifies the expected behavior for both Authorization Servers and MCP Clients when working with this pattern.

Motivation

MCP's authorization mechanism is based on OAuth 2.1, but many real-world deployments use Authorization Servers that also implement OpenID Connect (OIDC). A key difference between pure OAuth and OIDC is how refresh tokens are handled:

- In **pure OAuth 2.1**, there is no standard mechanism for a client to explicitly request a refresh token. The Authorization Server determines whether to issue one based on the client's capabilities (e.g., the `refresh_token` grant type in client metadata) and its own policies.
- In **OIDC** (and Authorization Servers that adopt this convention), the `offline_access` scope exists to allow clients to explicitly request refresh tokens, in addition to the OAuth logic.

This creates several problems in the MCP ecosystem:

1. **Clients aren't requesting refresh tokens**: Major MCP clients (Cursor, Claude, VS Code, etc.) aren't explicitly asking for refresh tokens via the `offline_access` scope because they don't know whether the Authorization Server supports, expects, or requires it.

2. **Resource servers shouldn't specify `offline_access`**: The `offline_access` scope is not a resource-specific scope—it's a concern between the client and Authorization Server. Including it in the `WWW-Authenticate` header's `scope` parameter or in the Protected Resource Metadata's `scopes_supported` would be semantically incorrect since it implies the resource *requires* refresh tokens, which it never would.

3. **Authorization Servers can be inconsistent**: When processing an authorization code grant, different Authorization Servers may have different behavior when issuing refresh tokens to different clients, especially when the client doesn't specify `refresh_token` as a grant type or request the `offline_access` scope.

4. **Interoperability gap**: Without this guidance, implementations may behave inconsistently, leading to poor user experience (frequent re-authentication) or security issues (issuing refresh tokens to clients that can't securely store them).

Specification

MCP Client Requirements

MCP Clients that intend to use refresh tokens and are capable of storing them securely **SHOULD** follow these guidelines:

1. **Advertise capability**: Clients **SHOULD** include `refresh_token` in their `grant_types` client metadata to indicate they support refresh tokens.
2. **Scope augmentation**: When the client desires a refresh token and the Authorization Server metadata contains `offline_access` in its `scopes_supported` field, the client **MAY** add the `offline_access` scope to the list of scopes from the resource server before making authorization requests to the Authorization Server.
3. **No guarantee**: Clients **MUST NOT** assume that advertising support or requesting `offline_access` guarantees they will receive a refresh token. The Authorization Server retains discretion based on its policies.

MCP Server (Resource Server) Requirements

MCP Servers (acting as OAuth 2.0 Protected Resources):

1. **SHOULD NOT** include `offline_access` in the `scope` parameter of `WWW-Authenticate` headers, as refresh tokens are not a resource requirement.
2. **SHOULD NOT** include `offline_access` in `scopes_supported` in Protected Resource Metadata, as it is not a resource-specific scope.

Rationale

Why not require `offline_access` in the 401 response?

The `offline_access` scope is fundamentally different from resource-specific scopes. It represents a client's desire for long-lived access, not a requirement of the resource. Per OAuth 2.1 Section 5.3.1, the `scope` attribute in `WWW-Authenticate` indicates "the required scope of the access token for accessing the requested resource." Since the resource doesn't require `offline_access`, including it would be semantically incorrect.

Why check client metadata for grant types? Why not always issue refresh tokens?

OAuth 2.1 requires clients to register their supported grant types. A client that doesn't support the `refresh_token` grant either:

- Cannot securely store refresh tokens
- Has no mechanism to use them

Issuing refresh tokens to such clients wastes Authorization Server resources (tracking tokens that will never be used) and may pose security risks if the tokens are leaked.

Why allow `offline_access` as an alternative signal?

Some Authorization Servers—whether fully OIDC-compliant or simply adopting this convention—only issue refresh tokens when `offline_access` is explicitly requested. Supporting this pattern provides a compatible path for such deployments. Clients can detect Authorization Servers that support this convention by checking for `offline_access` in `scopes_supported` in the Authorization Server Metadata and adapt their behavior accordingly.

Alternative approaches considered

1. **Mandate `offline_access` in resource responses**: Rejected because it misrepresents the resource's requirements and creates an anti-pattern.
2. **Always issue refresh tokens**: Rejected because it ignores client capabilities and Authorization Server security policies.
3. **Separate OIDC-specific specification**: Rejected in favor of a unified approach that works for both pure OAuth and OIDC deployments.
4. **Provide guidance for Authorization Servers**: Rejected in favor of relying on OAuth and OIDC specs for this guidance as it can vary.

Backward Compatibility

This proposal is fully backward-compatible:

- Clients that already request `offline_access` continue to work
- Authorization Servers that already check client capabilities continue to work
- MCP Servers are not required to make any changes
- The guidance is additive and does not change existing required behavior

Implementations that don't follow this guidance may experience suboptimal behavior (missing refresh tokens or unnecessary token issuance) but will remain functional.

Security Implications

Positive security implications

1. **Reduced token leakage risk**: By not issuing refresh tokens to clients that don't advertise support, we reduce the risk of long-lived tokens being stored insecurely.
2. **Defense in depth**: The risk-based assessment step gives Authorization Servers flexibility to implement additional security controls.

Considerations

1. **Client metadata may not be sufficient**: Since client metadata is self-reported, a malicious actor could register a client claiming `refresh_token` grant support to obtain long-lived tokens. Authorization Servers MAY use the risk-based assessment step (see Specification) to apply additional restrictions—such as domain allowlists, reputation checks, or verification requirements—rather than solely relying on client metadata claims when deciding whether to issue refresh tokens.
2. **Scope injection**: Clients adding `offline_access` should ensure this doesn't interfere with other scope-related logic or create unexpected authorization prompts.

Reference Implementation

Reference implementations demonstrating this guidance will be provided in the official MCP SDKs:

- **TypeScript SDK**: Client-side `offline_access` scope handling
- **Python SDK**: Client-side `offline_access` scope handling
- **Authorization Server example**: Demonstration of client capability checking
- **Client conformance test**: Allowing for easy validation of SDK implementations

Links to implementations will be added once the SEP is accepted.

Acknowledgments

This proposal was developed through discussion in the MCP Discord's authorization channel, with input from:

- Aaron Parecki (OAuth/OIDC expertise)
- Paul Carleton (MCP authorization guidance)
- Simon Russell (OIDC deployment experience)

SEP-2243 HTTP Header Standardization for Streamable HTTP Transport

Final · Standards Track · Created 2026-02-04

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-02-04
- **Author(s)**: MCP Transports Working Group
- **Sponsor**: None
- **PR**: https://github.com/modelcontextprotocol/specification/pull/2243

Abstract

This SEP proposes exposing critical routing and context information in standard HTTP header locations for the Streamable HTTP transport. By mirroring key fields from the JSON-RPC payload into HTTP headers, network intermediaries such as load balancers, proxies, and observability tools can route and process MCP traffic without deep packet inspection, reducing latency and computational overhead.

Motivation

Current MCP implementations over HTTP bury all routing information within the JSON-RPC payload. This creates friction for network infrastructure:

- **Load balancers** must terminate TLS and parse the entire JSON body to extract routing information (e.g., region, tool name)
- **Proxies and gateways** cannot make routing decisions without deep packet inspection
- **Observability tools** have limited visibility into MCP traffic patterns
- **Rate limiters and WAFs** cannot apply policies based on MCP-specific fields

By exposing key fields in HTTP headers, we enable standard network infrastructure to work with MCP traffic using existing, well-supported mechanisms.

Specification

Standard Headers

The Streamable HTTP transport will require POST requests to include the following headers mirrored from the request body:

Header Name	Source Field	Required For
`Mcp-Method`	`method`	All requests and notifications
`Mcp-Name`	`params.name` or `params.uri`	`tools/call`, `resources/read`, `prompts/get` requests

These headers are **required** for compliance with the MCP version in which they are introduced.

Server Behavior: Servers that process the request body MUST reject requests where the values specified in the headers do not match the values in the request body.

> **Rationale**: This requirement prevents potential security vulnerabilities and error conditions that could arise when different components in the network rely on different sources of truth. For example, a load balancer or gateway might use the header values to make routing decisions, while the MCP server uses the body values for execution. This requirement applies to any network intermediary that processes the message body, as well as the MCP server itself.

> **Implementation Note**: When validating integer parameter values, servers SHOULD compare the header value and the body value numerically rather than as strings (e.g., `42.0` and `42` are considered equal).

Case Sensitivity: Header names (called "field names" in RFC 9110) are case-insensitive. Clients and servers MUST use case-insensitive comparisons for header names.

Example: tools/call Request

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: tools/call
Mcp-Name: get_weather

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "Seattle, WA"
    }
  }
}
```

Example: resources/read Request

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: resources/read
Mcp-Name: file:///projects/myapp/config.json

{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "resources/read",
  "params": {
    "uri": "file:///projects/myapp/config.json"
  }
}
```

Example: prompts/get Request

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: prompts/get
Mcp-Name: code_review

{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "prompts/get",
  "params": {
    "name": "code_review",
    "arguments": {
      "language": "python"
    }
  }
}
```

Example: Other Request Methods

For requests that don't involve tools, resources, or prompts, only the `Mcp-Method` header is required:

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Method: initialize

{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "initialize",
  "params": {
    "protocolVersion": "2025-06-18",
    "capabilities": {},
    "clientInfo": {
      "name": "ExampleClient",
      "version": "1.0.0"
    }
  }
}
```

Example: Notification

Notifications also require the `Mcp-Method` header:

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: notifications/initialized

{
  "jsonrpc": "2.0",
  "method": "notifications/initialized"
}
```

Custom Headers from Tool Parameters

MCP servers MAY designate specific tool parameters to be mirrored into HTTP headers using an `x-mcp-header` extension property in the parameter's schema within the tool's `inputSchema`.

Client Requirement: While the use of `x-mcp-header` is optional for servers, clients MUST support this feature. When a server's tool definition includes `x-mcp-header` annotations, conforming clients MUST mirror the designated parameter values into HTTP headers as specified in this document.

Schema Extension

The `x-mcp-header` property specifies the name portion used to construct the header name `Mcp-Param-{name}`.

Constraints on `x-mcp-header` values:

- MUST NOT be empty
- MUST match HTTP field-name token syntax (`1*tchar`, RFC 9110 Section 5.1)
- MUST NOT contain control characters, including carriage return (CR, `\r`) or line feed (LF, `\n`)
- MUST be case-insensitively unique among all `x-mcp-header` values in the `inputSchema`

- MUST only be applied to parameters with primitive types (integer, string, boolean). Parameters with type `number` are not permitted. Integer values MUST be within the safe range for JavaScript (−2^53+1 to 2^53−1)
- MAY be applied to properties at any nesting depth within the `inputSchema`, not only top-level properties

Clients using the Streamable HTTP transport MUST reject tool definitions where any `x-mcp-header` value violates these constraints. Rejection means the client MUST exclude the invalid tool from the result of `tools/list`. Clients SHOULD log a warning when rejecting a tool definition, including the tool name and the reason for rejection. This behavior ensures that a single malformed tool definition does not prevent other valid tools from being used. Clients using other transports (e.g., stdio) MAY ignore `x-mcp-header` annotations entirely.

Example Tool Definition:

```
{
  "name": "execute_sql",
  "description": "Execute SQL on Google Cloud Spanner",
  "inputSchema": {
    "type": "object",
    "properties": {
      "region": {
        "type": "string",
        "description": "The region to execute the query in",
        "x-mcp-header": "Region"
      },
      "query": {
        "type": "string",
        "description": "The SQL query to execute"
      }
    },
    "required": ["region", "query"]
  }
}
```

Example: Geo-Distributed Database

Consider a server exposing an `execute_sql` tool for Google Cloud Spanner, which requires a `region` parameter.

Tool Definition:

```
{
  "name": "execute_sql",
  "description": "Execute SQL on Google Cloud Spanner",
  "inputSchema": {
    "type": "object",
    "properties": {
      "region": {
        "type": "string",
        "description": "The region to execute the query in",
        "x-mcp-header": "Region"
      },
      "query": {
        "type": "string",
        "description": "The SQL query to execute"
      }
    },
    "required": ["region", "query"]
  }
}
```

Scenario: A client requests to execute SQL in `us-west1`.

Current Friction: The global load balancer receives the request but must terminate TLS and parse the entire JSON body to find `"region": "us-west1"` before it knows whether to route the packet to the Oregon or Belgium cluster.

With This Proposal: The client detects the `x-mcp-header` annotation and automatically adds the header `Mcp-Param-Region: us-west1` to the HTTP request. The load balancer can now route based on the header without parsing the body.

Request:

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: tools/call
Mcp-Name: execute_sql
Mcp-Param-Region: us-west1

{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "execute_sql",
    "arguments": {
      "region": "us-west1",
      "query": "SELECT * FROM users"
    }
  }
}
```

Example: Multi-Tenant SaaS Application

A SaaS platform exposes tools that operate on different customer tenants. By exposing the tenant ID in a header, the platform can route requests to tenant-specific infrastructure.

Tool Definition:

```
{
  "name": "query_analytics",
  "description": "Query analytics data for a tenant",
  "inputSchema": {
    "type": "object",
    "properties": {
      "tenant_id": {
        "type": "string",
        "description": "The tenant identifier",
        "x-mcp-header": "TenantId"
      },
      "metric": {
        "type": "string",
        "description": "The metric to query"
      },
      "start_date": {
        "type": "string",
        "description": "Start date for the query range"
      },
      "end_date": {
        "type": "string",
        "description": "End date for the query range"
      }
    },
    "required": ["tenant_id", "metric", "start_date", "end_date"]
  }
}
```

Request:

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: tools/call
Mcp-Name: query_analytics
Mcp-Param-TenantId: acme-corp

{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "tools/call",
  "params": {
    "name": "query_analytics",
    "arguments": {
      "tenant_id": "acme-corp",
      "metric": "page_views",
      "start_date": "2026-01-01",
      "end_date": "2026-01-31"
    }
  }
}
```

Example: Priority-Based Request Handling

A server can expose a priority parameter to allow infrastructure to prioritize certain requests.

Tool Definition:

```
{
  "name": "generate_report",
  "description": "Generate a complex report",
  "inputSchema": {
    "type": "object",
    "properties": {
      "report_type": {
        "type": "string",
        "description": "Type of report to generate"
      },
      "priority": {
        "type": "string",
        "description": "Request priority: low, normal, or high",
        "x-mcp-header": "Priority"
      }
    },
    "required": ["report_type"]
  }
}
```

Request:

```
POST /mcp HTTP/1.1
Content-Type: application/json
Mcp-Session-Id: 1f3a4b5c-6d7e-8f9a-0b1c-2d3e4f5a6b7c
Mcp-Method: tools/call
Mcp-Name: generate_report
Mcp-Param-Priority: high

{
  "jsonrpc": "2.0",
  "id": 6,
  "method": "tools/call",
  "params": {
    "name": "generate_report",
    "arguments": {
      "report_type": "quarterly_summary",
      "priority": "high"
    }
  }
}
```

Header Processing

Value Encoding

Clients MUST encode parameter values before including them in HTTP headers to ensure safe transmission and prevent injection attacks.

Character Restrictions

Per RFC 9110, HTTP header field values must consist of visible ASCII characters (0x21-0x7E), space (0x20), and horizontal tab (0x09). The following characters are explicitly prohibited:

- Carriage return (`\r`, 0x0D)
- Line feed (`\n`, 0x0A)
- Null character (`\0`, 0x00)
- Any character outside the ASCII range (> 0x7F)

Whitespace Handling

HTTP parsers typically trim leading and trailing whitespace from header values. To preserve leading and trailing spaces in parameter values, clients MUST use Base64 encoding when the value:

- Starts with a space (0x20) or horizontal tab (0x09)
- Ends with a space (0x20) or horizontal tab (0x09)

Encoding Rules

Clients MUST apply the following encoding rules in order:

1. **Type conversion**: Convert the parameter value to its string representation:
    - `string`: Use the value as-is
    - `integer`: Convert to decimal string representation (e.g., `42`, `-7`)
    - `boolean`: Convert to lowercase `"true"` or `"false"`
2. **Whitespace check**: If the string starts or ends with whitespace (space or tab):

- Apply Base64 encoding (see below)

3. **ASCII validation**: Check if the string contains only valid ASCII characters (0x20-0x7E):
   - If valid, proceed to step 4
   - If invalid (contains non-ASCII characters), apply Base64 encoding (see below)
4. **Control character check**: If the string contains any control characters (0x00-0x1F or 0x7F):
   - Apply Base64 encoding (see below)

Base64 Encoding for Unsafe Values

When a value cannot be safely represented as a plain ASCII header value, clients MUST use Base64 encoding of the UTF-8 representation of the value with the following format:

```
Mcp-Param-{Name}: =?base64?{Base64EncodedValue}?=
```

The prefix `=?base64?` and suffix `?=` indicate that the value is Base64-encoded. These markers are case-sensitive and MUST appear exactly as shown (lowercase). Servers and intermediaries that need to inspect these values MUST decode them accordingly.

To avoid ambiguity, clients MUST also Base64-encode any plain-ASCII value that matches the sentinel pattern (i.e., starts with `=?base64?` and ends with `?=`).

Examples:

Original Value	Reason	Encoded Header Value
`"us-west1"`	Plain ASCII	`Mcp-Param-Region: us-west1`
`"Hello, 世界"`	Contains non-ASCII	`Mcp-Param-Greeting: =?base64?SGVsbG8sIOS4lueVjA==?=`
`" padded "`	Leading/trailing spaces	`Mcp-Param-Text: =?base64?IHBhZGRlZCA=?=`
`"line1\nline2"`	Contains newline	`Mcp-Param-Text: =?base64?bGluZTEKbGluZTI=?=`
`"=?base64?literal?="`	Matches sentinel pattern	`Mcp-Param-Val: =?base64?PT9iYXNlNjQ/bGl0ZXJhbD89?=`

Client Behavior

When constructing a `tools/call` request via HTTP transport, the client MUST:

1. Extract the values for any standard headers from the request body (e.g., `method`, `params.name`, `params.uri`)
2. Append the `Mcp-Method` header and, if applicable, `Mcp-Name` header to the request
3. Inspect the tool's `inputSchema` for properties marked with `x-mcp-header` and extract the value for each parameter
4. Encode the values according to the rules in Value Encoding
5. Append a `Mcp-Param-{Name}: {Value}` header to the request:

> **Implementation Note**: Clients MUST construct `Mcp-Param-*` headers using the most recently obtained `inputSchema` for the tool. A client that has never obtained the tool's `inputSchema` SHOULD send the request

> without `Mcp-Param-*` headers. If the server rejects the request because required `Mcp-Param-*` headers are missing or do not match the body, the client SHOULD call `tools/list` to obtain the current `inputSchema`, then retry the original request with the appropriate headers. Clients MAY pre-load tool definitions via other means (e.g., from a previous session or configuration) to enable header emission without a prior `tools/list` call.

Server Behavior

When receiving a request, the server MUST reject requests with `Mcp-Param-{Name}` headers that contain invalid characters (see "Character Restrictions" in the Value Encoding section).

Any server that processes the message body (not simply forwarding it) MUST validate that encoded header values, after decoding if Base64-encoded, match the corresponding values in the request body. Servers MUST reject requests with a `400 Bad Request` HTTP status if any validation fails.

Error Code

When rejecting a request due to header validation failure, servers MUST return a JSON-RPC error response with the following error code:

Code	Name	Description
`-32001`	`HeaderMismatch`	The HTTP headers do not match the corresponding values in the request body, or required headers are missing/malformed.

This error code is in the JSON-RPC implementation-defined server error range (`-32000` to `-32099`).

Error Response Format:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    "code": -32001,
    "message": "Header mismatch: Mcp-Name header value 'foo' does not match body value
'bar'"
  }
}
```

Validation Failure Conditions:

- A required standard header (`Mcp-Method` , `Mcp-Name` , etc.) is missing
- A header value does not match the request body value
- A Base64-encoded value cannot be decoded
- A header value contains invalid characters

> **Note**: Intermediaries MUST return an appropriate HTTP error status (e.g., `400 Bad Request`) for validation failures but are not required to return a JSON-RPC error response.

> **Note**: Intermediaries that enforce policy based on mirrored headers (e.g., routing or rate-limiting by tenant) SHOULD verify that the `MCP-Protocol-Version` header indicates a version that requires header–body

> validation. If the version is older or the header is absent, the intermediary SHOULD reject the request rather than trusting unvalidated header values.

Custom Header Handling:

Custom headers (those defined via `x-mcp-header`) follow the same validation rules as standard headers:

Scenario	Client Behavior	Server Behavior
Parameter value provided	Client MUST include the header	Server MUST validate header matches body
Parameter value is `null`	Client MUST omit the header	Server MUST NOT expect the header
Parameter not in arguments	Client MUST omit the header	Server MUST NOT expect the header
Client omits header but value is in body	Non-conforming client	Server MUST reject the request

When rejecting requests due to missing or invalid custom headers, the server MUST return HTTP status `400 Bad Request` with JSON-RPC error code `-32001` (`HeaderMismatch`).

Rationale

Headers vs Path

This proposal mirrors request data into headers rather than encoding it in the URL path.

Advantages of Headers:

1. **Simplicity**: All widely-used network load balancers support routing based on HTTP headers
2. **Multi-version support**: Easier to support multiple MCP versions in clients and servers
3. **Compatibility**: Headers work with the existing Streamable HTTP transport design without changing the endpoint structure
4. **Unlimited values**: Header values can contain characters that would require encoding in URLs (e.g., `/`, `?`, `#`)
5. **No URL length limits**: Very long values can be transmitted without hitting URL length restrictions

Advantages of Path-based Routing:

1. **Framework simplicity**: Many web frameworks (Flask, Express, Django, Rails) have built-in support for path-based routing with minimal configuration
2. **Logging**: URL paths are typically logged by default, making debugging easier

Trade-offs and Framework Considerations:

Framework	Header-based Routing	Path-based Routing
Flask (Python)	Requires middleware or decorators to extract headers before routing	Native support via `@app.route('/mcp/<method>')`
Express (Node.js)	Easy via `req.headers` but requires custom routing logic	Native support via `app.post('/mcp/:method')`
Django (Python)	Requires custom middleware	Native URL patterns
Go (net/http)	Easy via `r.Header.Get()`	Native via path patterns
ASP.NET Core	Easy via `[FromHeader]` attribute	Native via route templates

For frameworks like Flask that strongly favor path-based routing, implementing header-based routing requires additional code:

```
# Flask example: Header-based routing requires manual dispatch
@app.route('/mcp', methods=['POST'])
def mcp_handler():
    method = request.headers.get('Mcp-Method')
    if method == 'tools/call':
        return handle_tools_call(request)
    elif method == 'resources/read':
        return handle_resources_read(request)
    # ... etc
```

Despite this additional complexity in some frameworks, header-based routing was chosen because:

1. **Backwards Compatibility** introducing path based routing would require all existing MCP Servers to take a major update, and potentially support two sets of endpoints to support multiple versions. Even if the SDKs can paper over this additional operational concerns like testing, metrics, etc would need to happen. Header based routing requires minimal client side changes. And clients which don't opt in will still function correctly.
2. **Infrastructure benefits outweigh framework complexity**: The primary goal is enabling network infrastructure (load balancers, proxies, WAFs) to route and process requests without body parsing. This benefit applies regardless of the server framework.

Infrastructure Support

HTTP header-based routing and processing is supported by:

- **Load Balancers**: All major load balancers (HAProxy, NGINX, Cloudflare, F5, Envoy/Istio)
- **Rate Limiting**: 9 of 11 popular rate-limiting solutions
- **Authorization**: Kong, Tyk, AWS API Gateway, Google Cloud Apigee, Azure API Gateway, NGINX, Apache APISIX, Istio/Envoy
- **Web Application Firewalls**: Cloudflare WAF, AWS WAF, Azure WAF, F5 Advanced WAF, FortiWeb, Imperva WAF, Barracuda WAF, ModSecurity, Akamai, Wallarm
- **Observability**: Most observability solutions can extract data from HTTP headers

Explicit Header Names in x-mcp-header

The design uses an explicit name value in `x-mcp-header` rather than deriving the header name from the parameter name because:

1. **Case sensitivity mismatch**: Header names are case-insensitive, but JSON Schema property names are case-sensitive
2. **Character set constraints**: Header names are limited to ASCII characters, but tool parameter names may contain arbitrary Unicode
3. **Simplicity**: No complex scheme needed for constructing header names from nested properties

Placement Within JSON Schema

The `x-mcp-header` extension is placed directly within the JSON Schema of the property to be mirrored, rather than in a separate metadata field outside the schema. This design choice offers several advantages:

1. **Co-location**: The header mapping is defined alongside the property it affects, making it immediately clear which parameter will be mirrored. Developers don't need to cross-reference between the schema and a separate metadata structure.

2. **Established pattern**: JSON Schema explicitly supports extension keywords (properties starting with `x-`), and this pattern is widely used in ecosystems like OpenAPI. Tool authors and SDK developers are already familiar with this approach.

3. **Schema composability**: When schemas are composed, extended, or referenced using `$ref`, the `x-mcp-header` annotation travels with the property definition. A separate metadata structure would require complex synchronization logic to maintain consistency.

4. **Tooling compatibility**: Existing JSON Schema validators ignore unknown keywords by default, so adding `x-mcp-header` doesn't break existing schema validation. Tools that don't understand this extension simply skip it.

5. **Reduced complexity**: A separate metadata structure would require defining a mapping mechanism (e.g., JSON Pointer or property paths) to associate headers with properties, adding implementation complexity and potential for errors.

Scope: Tools Only

The `x-mcp-header` mechanism currently applies only to `tools/call` requests because tools are the only MCP primitive with an `inputSchema` that supports JSON Schema extension keywords. Resources and prompts lack an equivalent schema structure: `resources/read` takes only a `uri` (already exposed via `Mcp-Name`), and `prompts/get` defines arguments as a simple `{name, description, required}` array without JSON Schema extensibility. Generalizing custom header mapping to these primitives would require adding `inputSchema`-style definitions to resources and prompts, which is a larger specification change. This is noted as a potential future extension.

No Specification-Level Header Size Limit

This specification intentionally does not define limits on individual header value length, total MCP header size, or number of custom headers. Headers are solely an HTTP concept, and HTTP itself (RFC 9110) does not specify header size limits. Common HTTP infrastructure imposes its own limits — ranging from 4–8 KB on some servers (e.g., Apache at ~8190 bytes) to 128 KB on others (e.g., Cloudflare) — but the appropriate limit depends on the deployment environment, which only the service operator can determine.

Defining a specification-level limit (such as "omit headers exceeding 8192 bytes") would introduce problems:

1. **Arbitrary threshold**: Any chosen value would be too low for some deployments and irrelevant for others. The "right" limit varies by infrastructure.
2. **Counterproductive omission**: If a client omits a header because it exceeds a spec-defined limit, servers and intermediaries that rely on that header for routing must either parse the body or reject the request — undermining the core purpose of exposing values in headers.
3. **Unnecessary SDK burden**: SDK maintainers would need to implement and test limit-checking logic for a constraint that rarely applies in practice.
4. **Redundant with HTTP**: Servers and intermediaries already reject oversized headers using standard HTTP status codes (`413 Request Entity Too Large` , `431 Request Header Fields Too Large`), which clients must handle regardless.

> **Note to implementers**: Servers, intermediaries, and clients MAY independently impose limits on individual header size, total MCP header size, or number of custom headers as appropriate for their deployment environment. Servers SHOULD document any limits they impose. Clients SHOULD gracefully handle `413 Request Entity Too Large` or `431 Request Header Fields Too Large` responses. Tool authors SHOULD limit `x-mcp-header` annotations to parameters that provide clear infrastructure benefits.

Encoding Approach for Unsafe Values

Four approaches were considered for encoding parameter values that cannot be safely represented as plain ASCII header values (non-ASCII characters, leading/trailing whitespace, control characters):

1. **Sentinel wrapping (chosen approach)**: Use the `=?base64?{value}?=` prefix/suffix within the same `Mcp-Param-{Name}` header to signal Base64-encoded values.
2. **Separate header name**: Use a distinct header name for encoded values, e.g. `Mcp-ParamEncoded-{Name}` , so the encoding is indicated by the header name rather than the value format.
3. **Implicit encoding**: Let the parser infer encoding from the tool schema, e.g. via a `"x-mcp-header-encoding": "base64"` annotation in the tool definition.
4. **Always encode**: Base64-encode every `Mcp-Param-{Name}` value unconditionally.

Approach	Pros	Cons
Sentinel wrapping	Single header name per parameter; common case (plain ASCII) is human-readable; intermediaries can route on plain values without decoding	In-band signaling can theoretically collide with literal values; every reader must check for the prefix
Separate header name	No in-band ambiguity; encoding is self-documenting from the header name	Doubles the header namespace; every intermediary must check two header names per parameter; needs a conflict rule if both are present
Implicit encoding	Simplest wire format; no sentinels or extra headers	Intermediaries need access to the tool schema to know whether to decode — defeats the purpose of exposing values in headers; static per-parameter decision doesn't handle the mixed case well
Always encode	Simplest rules; no conditional logic or ambiguity	Plain ASCII values become unreadable; intermediaries must decode Base64 to inspect any value, significantly undermining the core motivation of this SEP

Conclusion: The sentinel wrapping approach provides the best trade-off. The primary use case for custom headers is enabling intermediaries to route and filter on simple, readable values like region names and tenant IDs — these are invariably plain ASCII and never trigger Base64 encoding. Option 4 makes all values opaque to intermediaries. Option 3 leaves intermediaries unable to distinguish encoded from literal values without access to the tool schema. Option 2 eliminates in-band ambiguity but doubles the header namespace, requiring intermediaries to check two possible header names per parameter and adding a conflict rule when both are present. The theoretical collision risk of the sentinel in Option 1 is negligible since `=?base64?...?=` is an unlikely literal parameter value in practice.

Backward Compatibility

Standard Headers

Existing clients and SDKs will be required to include the standard headers when using the new MCP version. This is a minor addition since clients already include headers like `Mcp-Protocol-Version`, adding only one or two new headers per message.

Servers implementing the new version MUST reject requests missing required headers. Servers MAY support older clients by accepting requests without headers when negotiating an older protocol version.

Custom Headers from Tool Parameters

The `x-mcp-header` extension is optional for servers. Existing tools without this property continue to work unchanged. However, clients implementing the MCP version that includes this specification MUST support the feature. Older clients that do not support `x-mcp-header` will still function but will not provide the header-based routing benefits that servers may depend on.

Security Implications

Header Injection

Header injection attacks occur when malicious values containing control characters (especially `\r\n`) are included in headers, potentially allowing attackers to inject additional headers or terminate the header section early.

Clients MUST follow the Value Encoding rules defined in this specification. These rules ensure that:

- Control characters are never included in header values
- Non-ASCII values are safely encoded using Base64
- Values exceeding safe length limits are omitted

Header Spoofing

Servers MUST validate that header values match the corresponding values in the request body. This prevents clients from sending mismatched headers to manipulate routing while executing different operations.

For example, a malicious client could attempt to:

- Route a request to a less-secured region while executing operations intended for a high-security region
- Bypass rate limits by spoofing tenant identifiers
- Evade security policies by misrepresenting the operation being performed

Information Disclosure

Tool parameter values designated for headers will be visible to network intermediaries (load balancers, proxies, logging systems). Server developers:

- SHOULD NOT mark sensitive parameters (passwords, API keys, tokens, PII) with `x-mcp-header`
- SHOULD document which parameters are exposed as headers
- SHOULD consider that Base64 encoding provides no confidentiality—it is merely an encoding, not encryption

Trusting Header Values

Header values originate from tool call arguments, which may be influenced by an LLM or a malicious client. Intermediaries and servers MUST NOT treat these values as trusted input for security-sensitive decisions. In particular:

- Header values that imply access to specific resources (e.g., tenant IDs, region names) MUST be independently verified against the authenticated user's permissions before granting access to those resources.
- Header values MUST NOT be used as the sole basis for granting elevated privileges without server-side enforcement of rate limits and quotas.
- Deployments SHOULD reject requests with oversized or excessive headers early in the pipeline — before performing Base64 decoding or body parsing — to mitigate denial-of-service risks from crafted payloads.

Conformance Test Cases

This section defines edge cases that conformance tests MUST cover to ensure interoperability between implementations.

Standard Header Edge Cases

Case Sensitivity

Test Case	Input	Expected Behavior
Header name case variation	`mcp-method: tools/call`	Server MUST accept (header names are case-insensitive)
Header name mixed case	`MCP-METHOD: tools/call`	Server MUST accept
Method value case	`Mcp-Method: TOOLS/CALL`	Server MUST reject (method values are case-sensitive)

Header/Body Mismatch

Test Case	Header Value	Body Value	Expected Behavior
Method mismatch	`Mcp-Method: tools/call`	`"method": "prompts/get"`	Server MUST reject with 400 and error code `-32001`
Tool name mismatch	`Mcp-Name: foo`	`"params": {"name": "bar"}`	Server MUST reject with 400 and error code `-32001`
Missing required header	(no `Mcp-Method`)	Valid body	Server MUST reject with 400 and error code `-32001`
Extra whitespace in header	`Mcp-Name: foo`	`"params": {"name": "foo"}`	Server MUST accept (trim whitespace per HTTP spec)

Special Characters in Values

Test Case	Value	Expected Behavior
Tool name with hyphen	`my-tool-name`	Client sends as-is; server accepts
Tool name with underscore	`my_tool_name`	Client sends as-is; server accepts
Resource URI with special chars	`file:///path/to/file%20name.txt`	Client sends as-is; server accepts
Resource URI with query string	`https://example.com/resource?id=123`	Client sends as-is; server accepts

Custom Header Edge Cases

x-mcp-header Name Conflicts

Test Case	Schema	Expected Behavior
Duplicate header names (same case)	Two properties with `"x-mcp-header": "Region"`	Client MUST reject tool definition
Duplicate header names (different case)	`"x-mcp-header": "Region"` and `"x-mcp-header": "REGION"`	Client MUST reject tool definition (case-insensitive uniqueness)
Header name matches standard header	`"x-mcp-header": "Method"`	Allowed (produces `Mcp-Param-Method`, not `Mcp-Method`)
Empty header name	`"x-mcp-header": ""`	Client MUST reject tool definition

Invalid x-mcp-header Values

Test Case	x-mcp-header Value	Expected Behavior
Contains space	`"x-mcp-header": "My Region"`	Client MUST reject tool definition
Contains colon	`"x-mcp-header": "Region:Primary"`	Client MUST reject tool definition
Contains non-ASCII	`"x-mcp-header": "Région"`	Client MUST reject tool definition
Contains control character	`"x-mcp-header": "Region\t1"`	Client MUST reject tool definition

Value Encoding Edge Cases

Test Case	Parameter Value	Expected Header Value
Plain ASCII string	`"us-west1"`	`Mcp-Param-Region: us-west1`
String with leading space	`" us-west1"`	`Mcp-Param-Region: =?base64?IHVzLXdlc3Qx?=`
String with trailing space	`"us-west1 "`	`Mcp-Param-Region: =?base64?dXMtd2VzdDEg?=`
String with leading/trailing spaces	`" us-west1 "`	`Mcp-Param-Region: =?base64?IHVzLXdlc3QxIA==?=`
String with internal spaces only	`"us west 1"`	`Mcp-Param-Region: us west 1`
Boolean true	`true`	`Mcp-Param-Flag: true`
Boolean false	`false`	`Mcp-Param-Flag: false`
Integer	`42`	`Mcp-Param-Count: 42`
Floating point	`3.14159`	`Mcp-Param-Value: 3.14159`
Non-ASCII characters	`"日本語"`	`Mcp-Param-Text: =?base64?5pel5pys6Kqe?=`
String with newline	`"line1\nline2"`	`Mcp-Param-Text: =?base64?bGluZTEKbGluZTI=?=`
String with carriage return	`"line1\r\nline2"`	`Mcp-Param-Text: =?base64?bGluZTENCmxpbmUy?=`
String with leading tab	`"\tindented"`	`Mcp-Param-Text: =?base64?CWluZGVudGVk?=`
Empty string	`""`	`Mcp-Param-Name:` (empty value)

Type Restriction Violations

Test Case	Property Type	x-mcp-header Present	Expected Behavior
Array type	`"type": "array"`	Yes	Server MUST reject tool definition
Object type	`"type": "object"`	Yes	Server MUST reject tool definition
Null type	`"type": "null"`	Yes	Server MUST reject tool definition
Nested property	Property inside object	Yes	Server MUST reject tool definition

Server Validation Edge Cases

Base64 Decoding

Test Case	Header Value	Expected Behavior
Valid Base64	`=?base64?SGVsbG8=?=`	Server decodes to `"Hello"` and validates
Invalid Base64 padding	`=?base64?SGVsbG8?=`	Server MUST reject with 400 and error code `-32001`; Intermediary MAY reject with 400 status code
Invalid Base64 characters	`=?base64?SGVs!!!bG8=?=`	Server MUST reject with 400 and error code `-32001`; Intermediary MAY reject with 400 status code
Missing prefix	`SGVsbG8=`	Server treats as literal value, not Base64
Missing suffix	`=?base64?SGVsbG8=`	Server treats as literal value, not Base64
Non-lowercase prefix	`=?BASE64?SGVsbG8=?=`	Server treats as literal value, not Base64

Null and Missing Values

Test Case	Scenario	Expected Behavior
Parameter with x-mcp-header is null	`"region": null`	Client MUST omit header
Parameter with x-mcp-header is missing	Parameter not in arguments	Client MUST omit header
Optional parameter present	Optional parameter provided	Client MUST include header

Missing Custom Header with Value in Body

Test Case	Header Present	Body Value	Expected Behavior
Standard header omitted, value in body	No `Mcp-Name`	`"params": {"name": "foo"}`	Server MUST reject with 400 and error code `-32001`; Intermediary MAY reject with 400 status code
Custom header omitted, value in body	No `Mcp-Param-Region`	`"region": "us-west1"`	Server MUST reject with 400 and error code `-32001`; Intermediary MAY reject with 400 status code

Reference Implementation

To be provided before this SEP reaches Final status.

Implementation requirements:

- **Server SDKs**: Provide a mechanism (attribute/decorator) for marking parameters with `x-mcp-header`
- **Client SDKs**: Implement the client behavior for extracting and encoding header values
- **Validation**: Both sides must validate header/body consistency

Changes since SEP became Final

This SEP is preserved as a historical record of what was accepted. The list below tracks changes made to the specification after this SEP reached Final status. Refer to the current specification for the authoritative, up-to-date requirements.

- **`HeaderMismatch` error code reassigned from `-32001` to `-32020`.** This SEP originally assigned `HeaderMismatch` to `-32001`. The error-code allocation update in #2907 reassigned `HeaderMismatch` to `-32020`. All references to `-32001` above should be read as `-32020` when implementing against the current specification.

SEP-2260 Require Server requests to be associated with a Client request.

Final · Standards Track · Created 2026-02-16

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-02-16
- **Author(s)**: MCP Transports Working Group
- **Sponsor**: @CaitieM20 - Caitie McCaffrey
- **PR**: https://github.com/modelcontextprotocol/specification/pull/2260

Abstract

This SEP clarifies that `roots/list`, `sampling/createMessage`, and `elicitation/create` requests **MUST** be associated with an originating client-to-server request (e.g., during `tools/call`, `resources/read`, or `prompts/get` processing). Standalone server-initiated requests of these types outside notifications **MUST NOT** be implemented.

Although not enforced in the current MCP Data Layer, logically these requests **MUST** be associated with a valid client-to-server JSON-RPC Request Id.

The operational server-to-client **Ping** is excepted from this restriction.

Motivation

Current Specification

The current specification uses **SHOULD** language in the transport layer:

In context of responding to a POST Request in the Streamable HTTP transport (2025-11-25/basic/transports.mdx:121-L123):

> - "The server **MAY** send JSON-RPC *requests* and *notifications* before sending the JSON-RPC *response*. These messages **SHOULD** relate to the originating client *request*."

For the optional GET SSE Stream (2025-11-25/basic/transports.mdx:146-L148):

> - "The server **MAY** send JSON-RPC *requests* and *notifications* on the stream."
> - "These messages **SHOULD** be unrelated to any concurrently-running JSON-RPC *request* from the client."

Although the GET stream allows "unsolicited" requests, its use is entirely optional and cannot be relied upon by MCP Server authors.

Design Intent

The design intent of MCP Server Requests is to operate reactively **nested within** other MCP operations:

- **Sampling** enables servers to request LLM assistance while processing a tool call, resource request, or prompt
- **Elicitation** enables servers to gather additional user input needed to complete an operation
- **List Roots** enables servers to identify shared storage locations

Ping has a special status as it is primarily intended as a keep-alive/health-check mechanism.

For Streamable HTTP Servers this enables SSE Streams to be maintained for extended periods if no Notifications or Requests are available to be sent. For client-to-server Requests they are associable. Future transport implementations will remove the need for dissociated Pings.

The current specification already describes this pattern:

> "Sampling in MCP allows servers to implement agentic behaviors, by enabling LLM calls to occur *nested* inside other MCP server features."

However, the normative requirements don't enforce this constraint.

Simplification Benefits

Making this constraint explicit:

1. **Simplifies transport implementations** - Transports don't need to support arbitrary server-initiated request/response flows, which require a persistent connection from Server to Client; they only need request-scoped bidirectional communication
2. **Clarifies user experience** - Users understand that sampling/elicitation happens *because* they initiated an action, not spontaneously
3. **Reduces security surface** - Ensures client has context for what scope the additional requested information will be used for. This allows clients to make better informed decisions on whether to provide the requested info.
4. **Aligns with practice** - Based on a scan of GitHub all existing implementations already follow this pattern, except one repo owned by the SEP author with a contrived scenario.

Specification Changes

1. Add Warning Blocks to Feature Documentation

In `client/sampling.mdx` (after existing security warning):

```
<Warning>

**Request Association Requirement**

Servers **MUST** send `sampling/createMessage` requests only in association with an
originating client request (e.g., during `tools/call`, `resources/read`, or
`prompts/get` processing).

Standalone server-initiated sampling on independent communication streams (unrelated to
any client request) is not supported and **MUST NOT** be implemented. Future transport
implementations are not required to support this pattern.

</Warning>
```

In `client/elicitation.mdx` (after existing security warning):

```
<Warning>

**Request Association Requirement**

Servers **MUST** send server-to-client requests (such as `roots/list`,
`sampling/createMessage`, or `elicitation/create`) only in association with an
originating client request (e.g., during `tools/call`, `resources/read`, or
`prompts/get` processing).

Standalone server-initiated requests of these types on independent
communication streams (unrelated to any client request) are not supported and
**MUST NOT** be implemented. Future transport implementations are not required
to support this pattern.

</Warning>
```

In `client/roots.mdx` (in `User Interaction Model` section):

```
<Warning>

Servers **MUST** send server-to-client requests (such as `roots/list`,
`sampling/createMessage`, or `elicitation/create`) only in association with an
originating client request (e.g., during `tools/call`, `resources/read`, or
`prompts/get` processing).

Standalone server-initiated requests of these types on independent
communication streams (unrelated to any client request) are not supported and
**MUST NOT** be implemented. Future transport implementations are not required
to support this pattern.

</Warning>
```

In `basic/utilities/ping.mdx` (In `Overview` section):

```
<Warning>

`ping` is an MCP-level liveness check and **MAY** be sent by either party at
any time on an established session/connection.

In Streamable HTTP, implementations **SHOULD** prefer transport-level SSE
keepalive mechanisms for idle-connection maintenance; `ping` remains available
for protocol-level responsiveness checks.

Request-association requirements for `roots/list`, `sampling/createMessage`,
and `elicitation/create` do not apply to `ping`.

</Warning>
```

2. Clarify Transport Layer Constraints

In `basic/transports.mdx`, POST-initiated SSE streams (line ~121):

```
- The server **MAY** send JSON-RPC _requests_ and _notifications_ before sending the
- JSON-RPC _response_. These messages **SHOULD** relate to the originating client
- _request_.
+ The server **MAY** send JSON-RPC _requests_ and _notifications_ before sending the
+ JSON-RPC _response_. These messages **MUST** relate to the originating client
+ _request_.
```

In `basic/transports.mdx`, GET-initiated standalone SSE streams (line ~147):

```
- The server **MAY** send JSON-RPC _requests_ and _notifications_ on the stream.
- These messages **SHOULD** be unrelated to any concurrently-running JSON-RPC
- _request_ from the client.
+ The server **MAY** send JSON-RPC _notifications_ and _pings_ on the stream.
+ These messages **SHOULD** be unrelated to any concurrently-running JSON-RPC
+ _request_ from the client, **except** that `roots/list`,
+ `sampling/createMessage`, and `elicitation/create` requests **MUST NOT** be
+ sent on standalone streams.
```

Backward Compatibility

Impact Assessment

This change is expected to have **minimal to no impact** on existing implementations:

1. **Common usage patterns are preserved** - Sampling/elicitation within tool execution, resource reading, and prompt handling remain fully supported
2. **No known implementations affected** - Research conducted on GitHub has shown only one implementation of this pattern. This singular implementation is owned by the SEP author.

What's Disallowed

The following pattern, which was never explicitly documented or recommended, is now explicitly prohibited:

```
# ❌ PROHIBITED: Standalone server push
async def background_task():
    while True:
        await asyncio.sleep(60)
        # Try to initiate sampling without any client request context
        await session.create_message(...)  # NOT ALLOWED
```

What Remains Supported

The canonical pattern remains fully supported:

```
# ✅ SUPPORTED: Sampling during tool execution
@mcp.tool()
async def analyze_data(data: str, ctx: Context) -> str:
    # Request LLM analysis while processing the tool call
    result = await ctx.session.create_message(
        messages=[SamplingMessage(role="user", content=...)]
    )
    return result.content.text
```

Implementation Guidance

For Server Implementers

No changes required if your server:

- Only uses server-to-client requests within tool handlers
- Only uses server-to-client requests within resource/prompt handlers
- Uses server-to-client requests synchronously as part of processing a client request

Changes required if your server:

- Attempts to initiate server-to-client requests on standalone HTTP GET streams
- Attempts to send server-to-client requests requests independent of client operations
- Has background tasks that try to invoke server-to-client requests

Alternative designs will need to be implemented for the "Changes Required" case.

Implementors performing unsolicited server-to-client requests (typically URL Elicitation) immediately following initialization are encouraged to lazily perform these requests within the scope of a client-to-server request that requires that information from the client.

Timeout Considerations

When an MCP Server initiates a "nested" request inside a client request, the duration of the parent request extends to include the user's response time.

Implementers **MUST** ensure that:

1. Transport timeouts (e.g. HTTP Request Timeout) are sufficient to accommodate "Human-in-the-loop" delays, which may be unbounded.
2. Short timeouts enforced by infrastructure (e.g. Load Balancers) may result in connection termination before the user responds. For Streamable HTTP, transport-level SSE keepalive mechanisms **SHOULD** be used to keep connections alive and reset timers; `ping` requests **MAY** additionally be used for protocol-level responsiveness checks.

For Client Implementers

No changes required - Clients should already handle sampling/elicitation requests in the context of their own outbound requests. Potential to simplify implementations if out-of-band is currently supported.

Clients receiving server-to-client requests with no associated outbound request **SHOULD** respond with a `-32602` (Invalid Params) error.

For Transport Implementers

Future transport implementations can rely on the guarantee that:

- Sampling/elicitation requests only occur within the scope of a client-initiated request
- Transports don't need to support arbitrary server-initiated request/response flows on standalone channels
- Request correlation and lifecycle management is simplified

Timeline

(This SEP intends to serve as a public notice of the change prior to future protocol versions that will not be compatible with this usage)

Alternatives Considered

1. Soft Deprecation

Use **SHOULD NOT** language to discourage but not prohibit the pattern.

Rejected because: The behavior was never intentionally supported, and leaving it ambiguous prevents transport simplification.

2. Keep Current Ambiguity

Leave the existing **SHOULD** language unchanged.

Rejected because: This blocks future transport implementations and leaves implementers uncertain about whether the pattern is supported.

3. Create a Capability Flag

Add a `sampling.standalone` or similar capability for servers that want this behavior.

Rejected because: This adds complexity for a use case with no known demand, and contradicts the "nested" design principle.

References

- Current sampling documentation: `/specification/draft/client/sampling.mdx`
- Current elicitation documentation: `/specification/draft/client/elicitation.mdx`
- Transport specification: `/specification/draft/basic/transports.mdx`
- User interaction model discussion in client concepts documentation

SEP-2322 Multi Round-Trip Requests

Final · Standards Track · Created 2026-02-03

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-02-03
- **Author(s)**: Mark D. Roth (@markdroth), Caitie McCaffrey (@CaitieM20), Gabriel Zimmerman (@gjz22)
- **Sponsor**: Caitie McCaffrey (@CaitieM20)
- **PR**: https://github.com/modelcontextprotocol/specification/pull/{2322}

Abstract

This proposal specifies a simple way to handle server-initiated requests in the context of a client-initiated request (e.g., an elicitation request in the context of a tool call) without requiring a shared storage layer shared across server instances or statefulness in load balancing, which will significantly reduce the cost of operating MCP servers at scale in the common case. It also reduces the HTTP transport's dependence on SSE streams, which cause problems in a lot of environments that cannot support long-lived connections.

This proposed way of handling server-initiated requests will replace the current approach of sending server-initiated requests. This is a breaking change.

This SEP also specifies the subset of client requests that a server can send a server-initiated request on. This is a reduced scope compared to the current spec and is also a breaking change.

Making a breaking change here is necessary since adoption of server-initiated request features like Elicitation, Sampling and ListRoots is very low or blocked for many Remote MCP servers or Server Hosted Clients due to the operational complextity of supporting the SSE streams and server-side state.

Motivation

Note: This SEP is intended to provide a generic mechanism for handling any server-initiated request in the context of any client-initiated request. For clarity, throughout this document, we will specifically discuss tool calls as a proxy for any client-initiated request, but it should be read as applying equally to (e.g.) resource or prompt requests; similarly, we will discuss elicitation requests as a proxy for any server-initiated request, but it should be read as applying equally to (e.g.) sampling requests.

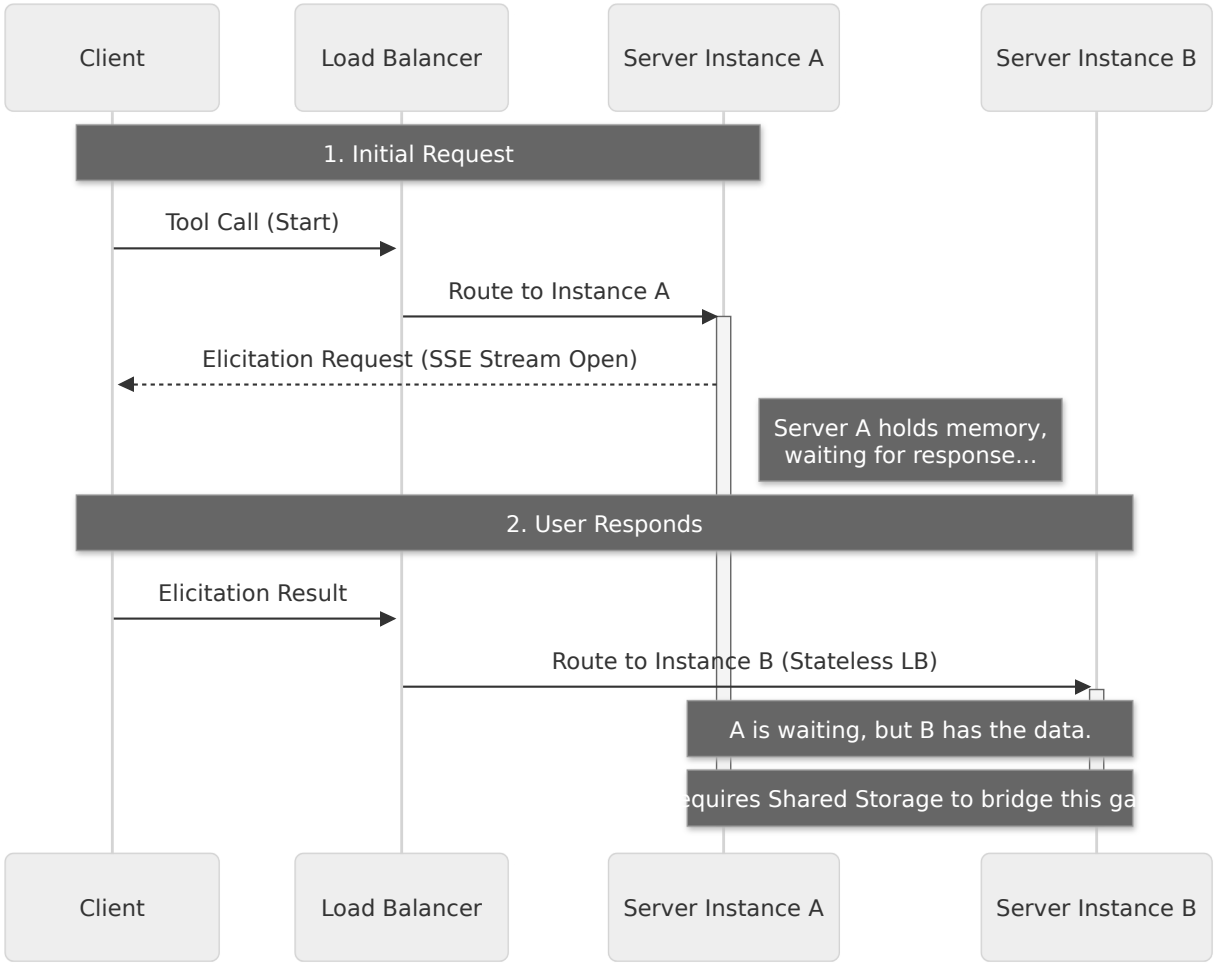
We start with the observation that there are two types of MCP tools:

1. **Ephemeral**: No state is accumulated on the server side.
    - If server needs more info to process the tool call, it can start from scratch when it gets that additional info.
    - Examples: weather app, accessing email
2. **Persistent**: State is accumulated on the server side.
    - Server may generate a large amount of state before requesting more info from the client, and it may need to pick up that state to continue processing after it receives the info from the client.
    - Server may need to continue processing in the background while waiting for more info from the client, in which case server-side state is needed to track that ongoing processing.
    - Examples: accessing an agent, spinning up a VM and needing user interaction to manipulate the VM

The vast majority of MCP tools will be ephemeral, and it is extremely common for tools to be deployed in a horizontally scaled, load balanced service, so we need to optimize for this case.

Today, if a tool needs to send an elicitation request in order to make progress, the workflow works like this:

1. Client sends tool call request. For this example, let's assume that the load balancers happen to send this request to server instance A.
2. Server A opens an SSE stream and sends the elicitation request on that stream.
3. Client sends the elicitation response as a separate request, for which the load balancers will choose a server instance completely independently of the one they chose in step 1. In this example, let's assume that the load balancers happen to send this request to server instance B.
4. Server A must somehow discover the elicitation response delivered to server B.
5. Server A then sends the tool call result on the SSE stream opened in step 2.



The difficult part here is step 4, which requires some sort of statefulness on the server side. The main way to solve this problem today is to have a storage layer shared across all server instances, so that multiple server instances can match up the elicitation response on one server instance with the original ongoing tool call on a different server instance.

There are two main approaches that can be used to solve this problem today:

- **Persistent Storage Layer Shared Across Server Instances**: Servers can deploy and manage a persistent storage layer (e.g., PostgreSQL, Redis, DynamoDB), which allow multiple server instances to match up the elicitation response on one server instance with the original ongoing tool call on a different server instance. This approach has a number of drawbacks:

    - The persistent storage layer is **extremely expensive**, especially for ephemeral tools that may not already have such a layer (e.g., a weather tool).
    - The persistent storage layer imposes significant reliability concerns: it becomes a critical dependency and therefore a potential single point of failure. To avoid that, it must provide high availability, replication, and backup mechanisms.
    - The persistent storage layer becomes a bottleneck, limiting horizontal scalability. Geographic distribution requires either expensive global replication or sticky routing.
    - The persistent storage layer also imposes significant operational complexity. In horizontally scaled deployments, it requires distributed locking or consensus protocols. It also requires special garbage collection logic to determine when shared can be cleaned up, which requires careful trade-offs: cleaning up state too aggressively can reduce storage costs but limit how long users have to respond, whereas cleaning up less aggressively accommodates slow users but increases storage costs.
    - This approach requires special behavior in the tool implementation to integrate with the persistent storage layer. The MCP SDKs today do not have any special hooks for this sort of storage layer integration, which means that it's very hard to write in-line code via the SDKs.
- **Statefulness in Load Balancing**: With the use of cookies, it is possible for the load balancing layer to ensure that the elicitation request in step 3 is delivered to the same server instance that the original request was delivered to in step 1. This approach, while often cheaper than a persistent storage layer, has the following drawbacks:
    - It requires special configuration and behavior in the load balancers, which is often difficult to manage.
    - It breaks normal load balancing models, resulting in uneven load distribution, thus increasing the cost of running the service.
    - It requires special behavior in clients to propagate the cookies used for statefulness.
    - It requires the tool implementation to match up the elicitation request with the ongoing tool call. (The MCP SDKs have some code to handle this, but it's still a very strange pattern in the HTTP world.)
    - It is not fault tolerant. If the server instance goes down, all state is lost, and the tool call would need to start over from scratch. (This doesn't necessarily matter for ephemeral tools, but it is an issue for persistent tools.)

Also, both of these approaches rely on the use of an SSE stream, which causes problems in environments that cannot support long-lived connections. They also require an instance of the tool to stay in memory in a particular server instance indefinitely. This is particularly problematic for elicitation requests specifically, since the result may not come from the user for an unbounded amount of time (e.g., it could be days or months, or maybe even never).

The goal of this SEP is to propose a simpler way to handle the pattern of server-initiated requests within the context of a client-initiated request. Specifically, we need to make it cheaper to support this pattern in the common case of an ephemeral tool in a horizontally scaled, load balanced deployment. This means that we need a solution that does not depend on an SSE stream and does not require either a persistent storage layer or stateful load balancing, which in turn means that we need to avoid dependencies between requests: servers must be able to process each individual request using no information other than what is present in that individual request.

Note that while the goal here is to optimize the common case of ephemeral tools, we do want to continue to support persistent tools, which generally already require a persistent storage layer.

Specification

This SEP proposes a new mechanism for handling server requests in the context of a client request. This new mechanism will have a slightly different workflow for ephemeral tools and persistent tools, the latter of which will leverage Tasks. However, both workflows will use the same data structures.

Schema Changes

First, we introduce the notion of `InputRequests`, which represents a set of one or more server-initiated request to be sent to the client, and `InputResponses`, which represents the client's responses to those requests. Both requests and responses are stored in a map with string keys. For `InputRequests`, the map values are server-initiated requests (e.g., elicitation or sampling requests), whereas for `InputResponses`, the map values are the responses to those requests. Here's how that would look in the typescript MCP schema:

```
export type InputRequest =
  CreateMessageRequest | ElicitRequest | ListRootsRequest;

export interface InputRequests {
  [key: string]: InputRequest;
}

export type InputResponse =
  CreateMessageResult | ElicitResult | ListRootsResult;

export interface InputResponses {
  [key: string]: InputResponse;
}
```

The keys are assigned by the server when issuing the requests. The client will send the response for each request using the corresponding key. For example, a server might send the following input requests:

```
"inputRequests": {
  // Elicitation request.
  "github_login": {
    "method": "elicitation/create",
    "params": {
      "mode": "form",
      "message": "Please provide your GitHub username",
      "requestedSchema": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string"
          }
        },
        "required": ["name"]
      }
    }
  },
  // Sampling request.
  "capital_of_france" : {
    "method": "sampling/createMessage",
    "params": {
      "messages": [
        {
          "role": "user",
          "content": {
            "type": "text",
            "text": "What is the capital of France?"
          }
        }
      ],
      "modelPreferences": {
        "hints": [
          {
            "name": "claude-3-sonnet"
          }
        ],
        "intelligencePriority": 0.8,
        "speedPriority": 0.5
      },
      "systemPrompt": "You are a helpful assistant.",
      "maxTokens": 100
    }
  }
}
```

The client would then send the responses in the following form:

```
"inputResponses": {
  // Elicitation response (ElicitResult).
  "github_login": {
    "action": "accept",
    "content": {
      "name": "octocat"
    }
  },
  // Sampling response (CreateMessageResult).
  "capital_of_france": {
    "role": "assistant",
    "content": {
      "type": "text",
      "text": "The capital of France is Paris."
    },
    "model": "claude-3-sonnet-20240307",
    "stopReason": "endTurn"
  }
}
```

The schema looks like this:

```
export interface InputRequiredResult extends Result {
  // Requests issued by the server that must be complete before the
  // client can retry the original request.
  inputRequests?: InputRequests;
  // Request state to be passed back to the server when the client
  // retries the original request.
  // Note: The client must treat this as an opaque blob; it must not
  // interpret it in any way.
  requestState?: string;
}

// RequestParams type that includes input responses and request state.
// These parameters may be included in any client-initiated request.
export interface InputResponseRequestParams extends RequestParams {
  // New field to carry the responses for the server's requests from the
  // InputRequiredResult message. For each key in the response's inputRequests
  // field, the same key must appear here with the associated response.
  inputResponses?: InputResponses;
  // Request state passed back to the server from the client.
  requestState?: string;
}
```

Since this change creates a polymorphic response for method calls like `tools/call`, we are introducing a new field to `Result` which indicate the `ResultType`. The client should parse this field to determine the type of the `Result` contained in the message. If this field is not provided, the Client should assume a `ResultType` of `"complete"` for backwards compatibility.

Extensions **MAY** add additional `ResultType` values. The set of supported `ResultType` values **MUST** be created from the set defined in the core protocol and include any additional values of supported extensions that are advertised via capabilities.

The Client **SHOULD** treat unrecognized values as invalid protocol responses.

The schema change will look like this:

```
/**
 * Common result fields.
 *
 * @category Common Types
 */
export interface Result {
  _meta?: MetaObject;
  // New field to indicate the type of the result, which allows the client to determine how to parse the result object. If no resultType is specified "complete" should be assumed.
  resultType: ResultType;
  [key: string]: unknown;
}

export type ResultType =
  | "complete" // the request completed successfully and the result contains the final content.
  | "input_required" // the request is incomplete and the result contains an {@link InputRequiredResult} object
  | string; // open to extensions
```

We anticipate this field will be useful for future extensibility, as it allows us to introduce new types of results and can also apply to `tasks` as well.

These types will be used in two different workflows, one for ephemeral tools and another for persistent tools.

Server-Initiated Request Support for Client Requests

Many `ClientRequest` don't have clear use cases where a Server would need to request more information from the Client. This SEP builds upon SEP-2260 and further restricts when a Server can send a Server-Initiated Request to the Client.

Servers MAY send `InputRequiredResult` responses on the following Client Requests:

ClientRequest	ServerResult	InputRequiredResult Supported
`GetPromptRequest`	`GetPromptResult`	Yes
`ReadResourceRequest`	`ReadResourceResult`	Yes
`CallToolRequest`	`CallToolResult`	Yes
`GetTaskPayloadRequest`	`GetTaskPayloadResult`	Yes

Servers MUST NOT send `InputRequiredResult` responses on any other Client Requests. The below table represents what `ClientRequest`s this excludes at the writing of this SEP.

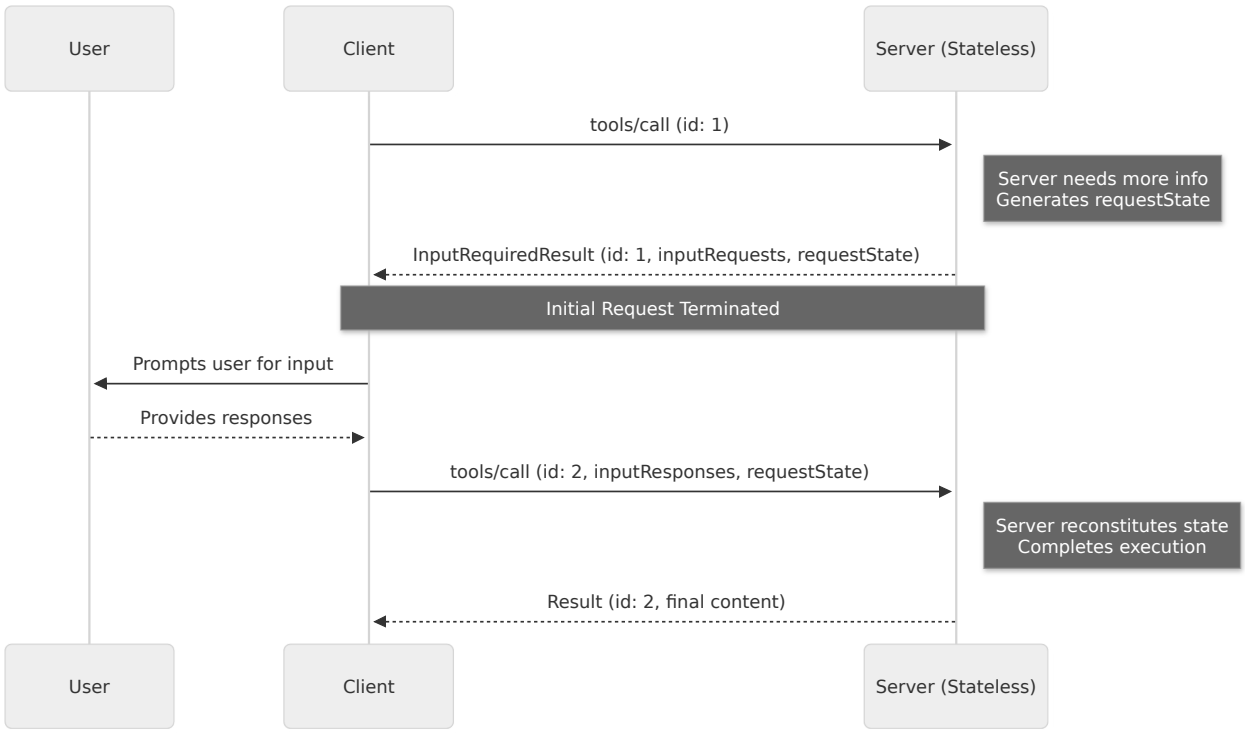
ClientRequest	InputRequiredResult Supported
`PingRequest`	No
`InitializeRequest`	No
`CompleteRequest`	No
`SetLevelRequest`	No
`ListPromptsRequest`	No
`ListResourcesRequest`	No
`ListResourceTemplatesRequest`	No
`SubscribeRequest`	No
`UnsubscribeRequest`	No
`ListToolsRequest`	No
`GetTaskRequest`	No
`ListTasksRequest`	No
`CancelTaskRequest`	No
`TaskInputResponseRequest`	No

Ephemeral Tool Workflow

For the ephemeral use case, in addition to input requests, we introduce the concept of request state. In cases where the server needs more information, the request state is sent to the client which echoes back the state to the server, allowing the server to remain stateless.

We will adopt the following workflow for ephemeral tools:

1. Client sends tool call request.
2. Server sends back a single response indicating that the request is incomplete. The response may include input requests that the client must complete. It may also include some request state that the client must return back to the server. This response terminates the original request. It will normally be sent as a single response, not on an SSE stream, although for now (this may change in a future SEP) it is also legal to send this response on an SSE stream following (e.g.) progress notifications. If this incomplete response is sent on an SSE stream, it must be the last message on the SSE stream, just as if it were a normal response.
3. Client sends a new tool call request, completely independent of the original one. This new tool call includes responses to the input requests from step 2. It also includes the request state specified by the server in step 2.
4. Server sends back a CallToolResponse.



Note that the requests in steps 1 and 3 are completely independent: the server that processes the request in step 3 does not need any information that is not directly present in the request. To support this decoupling the JsonRPC Id MUST be different between the requests sent in step 1 and step 3.

Note that both the "inputRequests" and "requestState" fields affect only the client's next retry of the original request. They will not be used for any other request that the client may be sending in parallel (e.g., a tool list or even another tool call).

▼ Click to expand Example Flow for Ephemeral Tools

Example Flow for Ephemeral Tools

Note: This is a contrived example, just to illustrate the flow.

1. The client sends the initial call tool request:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    }
  }
}
```

2. The server responds with an incomplete response, indicating that the client needs to respond to an elicitation request in order for the tool call to complete, and including request state to be passed back:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "github_login": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please provide your GitHub username",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "name": {
                "type": "string"
              }
            },
            "required": ["name"]
          }
        }
      }
    },
    "requestState": "foo"
  }
}
```

3. The client then retries the original tool call, this time including the responses to the input server request and the request state:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    },
    "inputResponses": {
      "github_login": {
        "action": "accept",
        "content": {
          "name": "octocat"
        }
      }
    },
    "requestState": "foo"
  }
}
```

4. Finally, the server completes the tool call:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "Current weather in New York:\nTemperature: 72°F\nConditions: Partly
cloudy"
      }
    ],
    "isError": false
  }
}
```

Real-World Example for Ephemeral Workflow

This example demonstrates how `requestState` enables a multi-round-trip elicitation flow driven by Azure DevOps custom rules. The scenario involves an `update_work_item` tool that transitions a Bug work item to "Resolved." ADO custom rules require specific fields when certain state transitions occur, and the server uses iterative elicitation to gather them — accumulating context in `requestState` across rounds so that the final update can be executed without any server-side storage.

▼ Click to expand ADO Custom Rules Example

Background — ADO Custom Rules in effect:

- *Rule 1:* When State changes to "Resolved" → require the "Resolution" field (e.g., Fixed, Won't Fix, Duplicate, By Design).
- *Rule 2:* When Resolution is "Duplicate" → require the "Duplicate Of" field (a link to the original work item).

Round 1 — Tool call triggers state change, server elicits Resolution

1. The client invokes the `update_work_item` tool to resolve Bug #4522:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "update_work_item",
    "arguments": {
      "workItemId": 4522,
      "fields": { "System.State": "Resolved" }
    }
  }
}
```

2. The server recognizes that setting State to "Resolved" triggers Rule 1, which requires a Resolution value. Rather than failing the call, the server returns an incomplete response with an elicitation request. No `requestState` is needed yet, since the original tool call arguments will be re-sent on retry:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "resolution": {
        "method": "elicitation/create",
        "params": {
          "message": "Resolving Bug #4522 requires a resolution. How was this bug
resolved?",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "resolution": {
                "type": "string",
                "enum": ["Fixed", "Won't Fix", "Duplicate", "By Design"],
                "description": "Resolution type for this bug"
              }
            },
            "required": ["resolution"]
          }
        }
      }
    }
  }
}
```

3. The user selects "Duplicate". The client retries the original tool call with the elicitation response:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "update_work_item",
    "arguments": {
      "workItemId": 4522,
      "fields": { "System.State": "Resolved" }
    },
    "inputResponses": {
      "resolution": {
        "action": "accept",
        "content": { "resolution": "Duplicate" }
      }
    }
  }
}
```

Round 2 — Resolution triggers another rule, server elicits Duplicate Of

4. The server merges the user's response and sees that Resolution = "Duplicate" triggers Rule 2, requiring a "Duplicate Of" link. It returns another incomplete response, this time encoding the already-gathered resolution in `requestState` so it is available regardless of which server instance handles the next retry:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "duplicate_of": {
        "method": "elicitation/create",
        "params": {
          "message": "Since this is a duplicate, which work item is the original?",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "duplicateOfId": {
                "type": "number",
                "description": "Work item ID of the original bug"
              }
            },
            "required": ["duplicateOfId"]
          }
        }
      }
    },
    "requestState": "eyJyZXNvbHV0aW9uIjoiRHVwbGljYXRlIn0..."
  }
}
```

5. The user provides the original work item ID. The client retries the tool call, echoing back the `requestState` and including the new elicitation response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "update_work_item",
    "arguments": {
      "workItemId": 4522,
      "fields": { "System.State": "Resolved" }
    },
    "inputResponses": {
      "duplicate_of": {
        "action": "accept",
        "content": { "duplicateOfId": 4301 }
      }
    },
    "requestState": "eyJyZXNvbHV0aW9uIjoiRHVwbGljYXRlIn0..."
  }
}
```

Final — Server completes the update

6. The server decodes the `requestState` (which contains the resolution), reads the `inputResponses` (which contains the duplicate ID), and now has all required fields. It completes the tool call:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "resultType": "complete",
    "content": [
      {
        "type": "text",
        "text": "Bug #4522 resolved as Duplicate of Bug #4301. State set to Resolved and
duplicate link created."
      }
    ],
    "isError": false
  }
}
```

Key takeaway: Across both elicitation rounds, the server held no in-memory or persisted state. The `requestState` field carried the accumulated context through the client, and any server instance could have handled any individual round.

Use Cases for Request State

The "requestState" mechanism provides a mechanism for doing multiple round trips on the same logical request. There are two main use-cases for this.

Use Case 1: Rolling Upgrades

Let's say that you are doing a rolling upgrade of your horizontally scaled server instances to deploy a new version of a tool implementation. The old version had two input requests with keys "github_login" and "google_login". However, in the new version of the tool implementation, it still uses the "github_login" input request, but it replaces the "google_login" input request with a new "microsoft_login" input request.

If the first request goes to an old version of the server but the second attempt (that includes the input responses) goes to a new version of the server, then the server will see the result for "github_login", which it needs, but it won't see the result for "microsoft_login". (It will also see the result for "google_login", but it no longer needs that, so it doesn't matter.) At this point, the server needs to send a new input request for "microsoft_login", but it also doesn't want to lose the answer that it's already gotten for "github_login", so it would use the kind of state proposed in 1685 to retain that information without having to store the state on the server side.

The workflow here would look like this:

1. Client sends tool call request that hits a server instance running the old version.
2. Server sends back an incomplete response indicating the input requests for "github_login" and "google_login".
3. Client sends a new tool call request that includes the responses to the input requests for "github_login" and "google_login". This time it hits a server instance running the new version.
4. Server sends back another incomplete response indicating the input request for "microsoft_login", which the client has not already provided. However, the response also includes request state containing the already-provided "github_login" response, so that the client does not need to prompt the user for the same information a second time.
5. Client sends a third tool call request that includes the response to the "microsoft_login" input request as well as echoing back the request state provided by the server in step 4.
6. Server now sees the "github_login" info in the request state and the "microsoft_login" state in the input responses, so the request now contains everything the server needs to perform the tool call and send back a complete response.

Use Case 2: Load Shedding

Let's say that you have an MCP server instance that is processing a bunch of tool calls and notices that it's too heavily loaded, so it wants to move one of the ongoing tool calls to a different server instance. However, it has already done a significant amount of processing on that tool call, so it does not want to simply fail the call and have the client start over from scratch on another server instance; instead, it wants to preserve the state it has already accumulated, so that whichever server instance resumes processing can pick up from where the original server instance left off. This can be accomplished by sending an incomplete request that contains request state but does not contain any input requests.

The workflow here would look like this:

1. Client sends the original request, which the load balancers route to server instance A.
2. Server instance A does a bunch of computation before deciding that it needs to shed load. It sends an incomplete response with its accumulated state in the `requestState` field but without the `inputRequests` field.
3. Client retries the request with the `requestState` field attached. The load balancers route this request to server instance B.
4. Server instance B starts from the state it sees in the `requestState` field, thus picking up the computation from where server instance A left off, and eventually returning a complete response.

Protocol Requirements for Ephemeral Workflow

1. **Server Behavior:**

- Servers MAY respond to any client-initiated request with a `InputRequiredResult`. This message MAY be sent either as a standalone response or as the final message on an SSE stream, although implementations are encouraged to prefer the former. If using an SSE stream, servers MUST NOT send any message on the stream after the incomplete response message.
- The `InputRequiredResult` MAY include an `inputRequests` field.
- The `InputRequiredResult` MAY include a `requestState` field. If specified, this field is an opaque string that is meaningful only to the server. Servers are free to encode the state in any format (e.g., plain JSON, base64-encoded JSON, encrypted JWT, serialized binary, etc.).
- If a request contains a `requestState` field, servers MUST always validate that state, as the client is an untrusted intermediary. If tampering is a concern, servers SHOULD encrypt the `requestState` field using an encryption algorithm of their choice (e.g., they can use AES-GCM or a signed JWT) to ensure both confidentiality and integrity. Note that there is also a risk of replaying/hijacking attacks, where an authenticated attacker resends state that was originally sent to a different user. Therefore, if the request state contains any data that is specific to the original user, the server MUST use some mechanism to cryptographically bind the data to the original user and MUST verify that the `requestState` data sent by the client is associated with the currently authenticated user. Servers using plaintext state MUST treat the decoded values as untrusted input and validate them the same way they would validate any client-supplied data.

2. **Client Behavior:**

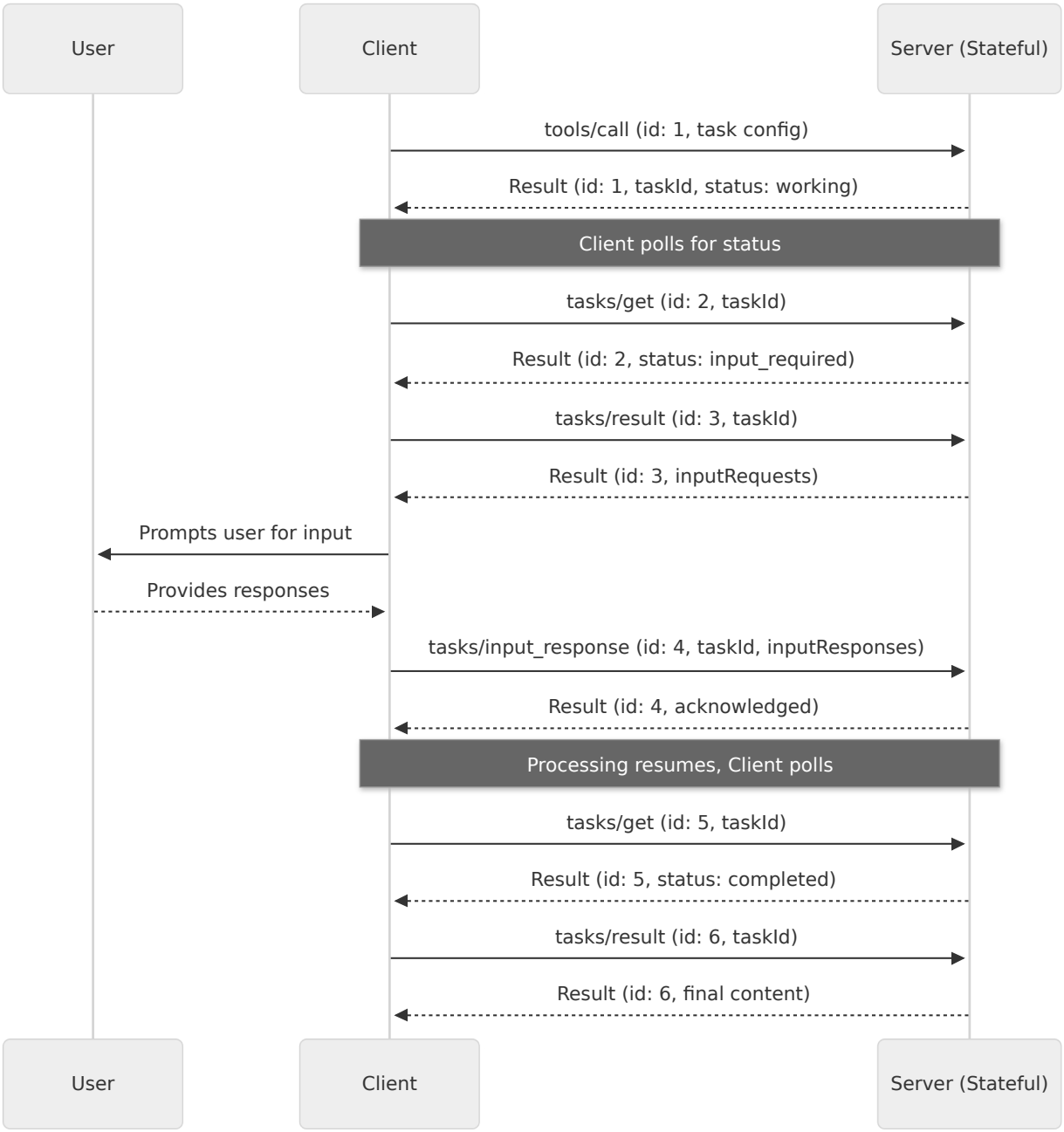
- If a client receives an `InputRequiredResult` message, if the message contains the `inputRequests` field, then the client MUST construct the requested input before retrying the original request. In contrast, if the message does *not* contain the `inputRequests` field, then the client MAY retry the original request immediately.
- If a client receives a `InputRequiredResult` message that contains the `requestState` field, it MUST echo back the exact value of that field when retrying the original request. Clients MUST NOT inspect, parse, modify, or make any assumptions about the `requestState` contents. If the `InputRequiredResult` does not contain a `requestState` field, the client MUST NOT include one in the retry.

Persistent Tool Workflow

The persistent tool workflow will leverage Tasks. `Tasks` already provide a mechanism to indicate that more information is needed to complete the request. The `input_required` Task Status allows the server to indicate that additional information is needed to complete processing the task.

The workflow for `Tasks` is as follows:

1. Server sets Task Status to `input_required`. The server can pause processing the request at this point.
2. Client retrieves the Task Status by calling `tasks/get` and sees that more information is needed.
3. Client calls `tasks/result`
4. Server returns the `InputRequests` object.
5. Client calls `tasks/input_response` request that includes an `InputResponses` object along with `Task` metadata field.
6. Server resumes processing sets TaskStatus back to `working`.



Since `Tasks` are likely longer running, have state associated with them, and are likely more costly to compute, the request for more information does not end the originally requested operation (e.g., the tool call). Instead, the server can resume processing once the necessary information is provided.

To align with MRTR semantics, the server will respond to the `tasks/result` request with a `InputRequests` object. Both of these will have the same JsonRPC `id`. When the client responds with a `InputResponses` object this is a new client request with a new JSONRPC `id` and therefore needs a new method name. We propose `tasks/input_response`.

The above workflow and below example do not leverage any of the optional Task Status Notifications although this SEP does not preclude their use.

▼ Click to expand Example Flow for Persistent Tools

The below example walks through the entire Task Message flow for a Echo Tool which can request additional information from the client via Elicitation.

1. **Client Request** to invoke EchoTool.

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "echo",
    "task": {
      "ttl": 60000
    }
  }
}
```

2. **Server Response** with a `Task`

```
{
  "id": 1,
  "jsonrpc": "2.0",
  "result": {
    "task": {
      "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5",
      "status": "working",
      "statusMessage": "Task has been created for echo tool invocation.",
      "createdAt": "2026-01-27T03:32:48.3148180Z",
      "lastUpdatedAt": "2026-01-27T03:32:48.3148180Z",
      "ttl": 60000,
      "pollInterval": 100
    }
  }
}
```

3. **Client Request** periodically checks the status of the `Task` using `tasks/get`.

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tasks/get",
  "params": {
    "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
  }
}
```

4. **Server Response** with Task status `input_required`

```
{
  "id": 2,
  "jsonrpc": "2.0",
  "result": {
    "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5",
    "status": "input_required",
    "statusMessage": "Input Required to Proceed call tasks/result",
    "createdAt": "2026-01-27T03:38:07.7534643Z",
    "lastUpdatedAt": "2026-01-27T03:38:07.7534643Z",
    "ttl": 60000,
    "pollInterval": 100
  }
}
```


5. **Client Request** sends message `tasks/result` to discover what input is required to proceed.


```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tasks/result",
  "params": {
    "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
  }
}
```


6. **Server Response** returns `inputRequests` to request additional input

```
{
  "id": 3,
  "jsonrpc": "2.0",
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "echo_input": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please provide the input string to echo back",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "input": { "type": "string" }
            },
            "required": ["input"]
          }
        }
      }
    }
  },
  "_meta": {
    "io.modelcontextprotocol/related-task": {
      "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
    }
  }
}
```

7. **Client Request** presents the Elicitation to the user and collects the input, then sends message to the server.

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tasks/input_response",
  "params": {
    "inputResponses": {
      "echo_input": {
        "action": "accept",
        "content": {
          "input": "Hello World!"
        }
      }
    },
    "_meta": {
      "io.modelcontextprotocol/related-task": {
        "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
      }
    }
  }
}
```

8. **Server Response** Server should acknowledge the receipt of the 'tasks/input_response' message by sending a 'JSONRPCResponse'. If the message was successfully received a `JSONRPCResultResponse` is sent including the `taskId`. If an error occurs, a `JSONRPCErrorResponse` is sent. The server can now proceed to complete the `Task` using the provided input, and the `Task` status changes to `Working`.

```
{
  "id": 4,
  "jsonrpc": "2.0",
  "result": {
    "_meta": {
      "io.modelcontextprotocol/related-task": {
        "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
      }
    }
  }
}
```

9. **Client Request** continues to poll the input status using `tasks/get` until server responds with Task Status of `Completed`

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "tasks/get",
  "params": {
    "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
  }
}
```

10. **Server Response** with Task status `completed`

```
{
  "id": 5,
  "jsonrpc": "2.0",
  "result": {
    "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5",
    "status": "completed",
    "statusMessage": "Task has been completed successfully, call tasks/result",
    "createdAt": "2026-01-27T03:38:07.7534643Z",
    "lastUpdatedAt": "2026-01-27T03:38:08.1234567Z",
    "ttl": 60000,
    "pollInterval": 100
  }
}
```

11. **Client Request** calls `tasks/result` to get the final result of the `Task` from the server.

```
{
  "id": 6,
  "jsonrpc": "2.0",
  "method": "tasks/result",
  "params": {
    "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
  }
}
```

12. **Server Response** with the final result of the `Task`

```
{
  "id": 6,
  "jsonrpc": "2.0",
  "result": {
    "resultType": "complete",
    "isError": false,
    "content": [
      {
        "type": "text",
        "text": "Echo: Hello World!"
      }
    ],
    "_meta": {
      "io.modelcontextprotocol/related-task": {
        "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
      }
    }
  }
}
```

Protocol Requirements for Persistent Workflow

1. **Server Behavior:**
   - Servers MAY respond to `tasks/get` by indicating that the task is in state `input_required`.
   - Servers MUST include an `inputRequests` field in the `tasks/result` response when the task is in state `input_required`.
2. **Client Behavior:**
   - When `tasks/get` shows state `input_required`, clients MUST call `tasks/result` to get the input requests. Clients SHOULD construct the results of those requests, and then call `tasks/input_response` with the input responses to provide the required input for the task.
   - Clients MAY choose not to fulfill the input requests, in which case they can cancel the task.

Interactions Between Ephemeral and Persistent Workflows

If a tool implementation needs the client to respond to a set of input requests before it can even start processing but then later needs to do persistent processing, it can start using the ephemeral workflow and then switch to the persistent workflow by creating a task at that point. This avoids the need for the server to store state until it actually has the information needed to start processing the request. This workflow would look like this:

1. Client sends tool call request with task metadata.
2. Server sends back `inputRequests` response indicating that more information is needed to process the request. This terminates the original request.
3. Client sends a new tool call request, completely independent of the original one, which includes the `inputResponses` object along with the task metadata.
4. Server sends back a task ID, indicating that it will be processing the request in the background. All subsequent interaction will be done via the Tasks API.

Note that the opposite is not true: Once a tool implementation returns a task, it has committed to storing state on the server side for the duration of the task, and there is no way to transition back to the ephemeral model. All subsequent interactions must be performed via the Tasks API.

Guidance for Error Handling

This section provides implementation guidance for error handling in scenarios where the client provides unexpected or malformed data in the `inputResponses` object.

As with any received request, the server SHOULD validate the data provided by the client is a valid `inputResponses` object and that the information inside can be correctly parsed. Protocol errors, like malformed JSON, invalid schema, or internal server errors which prevent the processing of the request should return a `JSONRPCErrorResponse` with an appropriate error code and message.

If additional parameters are provided in the `inputResponses` object The server SHOULD treat these as optional parameters. Therefore it SHOULD ignore any unexpected information in the `inputResponses` object that it does not recognize or need.

The client may also fail to send all the information requested in previous `inputRequests`. If the missing information requested is necessary for the server to process the request, then it SHOULD respond with a new `InputRequiredResult`.

We discussed having a specific application level error code returned, however the client may not have enough information to recover in all scenarios. Therefore, we decided to rely on the existing mechanics of requesting more input via `InputRequiredResult` to ensure a client can always recover by having the server request the necessary information again.

Malicious clients could intentionally send incorrect information in the `inputResponses` object, and generate load on the server by causing it to repeatedly request the same information. However, this is not a new concern introduced by this workflow, since malicious clients could already generate load by sending malformed requests. Server implementors can use standard techniques like rate limiting and throttling to protect themselves from such attacks.

In the ephemeral workflow, this would look like the following:

1. The client retries the original tool call, this time including the `inputResponses` object, but the response is missing required information that the server needs to process the request.

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "location": "New York"
    },
    "inputResponses": {
      "not_requested_info": {
        "action": "accept",
        "content": {
          "not_requested_param_name": "Information the server did not request"
        }
      }
    }
  }
}
```

2. The server responds with an incomplete response, indicating that the client needs to respond to an elicitation request in order for the tool call to complete, and including request state to be passed back:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "input_required",
    "inputRequests": {
      "github_login": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please provide your GitHub username",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "name": {
                "type": "string"
              }
            },
            "required": ["name"]
          }
        }
      }
    }
  }
}
```

2. The Server responds with an incomplete response, indicating that the client needs to provide missing information for the request to succeed.

In the persistent workflow, this would look like the following: Step 7 from above: **Client Request** The client mistakenly or maliciously sends unexpected, but well-formed data to the server in response to the input request.

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tasks/input_response",
  "params": {
    "inputResponses": {
      "echo_input": {
        "action": "accept",
        "content": {
          "not_requested_parameter": "Information the server did not request."
        }
      }
    },
    "_meta": {
      "io.modelcontextprotocol/related-task": {
        "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
      }
    }
  }
}
```

Step 8 from above. **Server Response** Server acknowledges the receipt of the response by sending a `JSONRPCResultResponse`. However, since the response is missing required information, the server does not proceed with processing the task and leaves the Task status as `input_required`. The next time the client calls `tasks/result`, the server responds with a new `inputRequest` requesting the necessary information again.

```
{
  "id": 4,
  "jsonrpc": "2.0",
  "result": {
    "_meta": {
      "io.modelcontextprotocol/related-task": {
        "taskId": "echo_dc792e24-01b5-4c0a-abcb-0559848ca3c5"
      }
    }
  }
}
```

Rationale

We considered a bidirectional stream approach to replace SSE streams. However, that approach would have made the wire protocol more complicated (e.g., it would have required HTTP/2 or HTTP/3). Also, it would not have eliminated problems for environments that cannot support long-lived connections, nor would it have addressed fault tolerance issues.

There was discussion about whether the input requests should be a map or just a single object, possibly leveraging some field inside of the requests (e.g., the elicitation ID) to differentiate between them. We decided that the map makes sense, since it structurally guarantees the uniqueness of keys, which will avoid the need for explicit checks in SDKs and applications to avoid conflicts.

In the persistent workflow, we considered including the input requests directly in the `tasks/get` response, rather than requiring the client to see the `input_required` status and then call `tasks/result` to get the input requests. We decided to keep those two things separate in deference to implementations that use separate infrastructure for task state and for the actual tool implementation; the idea is that the `tasks/get` call should have a consistent latency profile, regardless of what the task state actually is. We recognize that this requires an extra round-trip to the server, but we can optimize this in the future if becomes a problem.

Backward Compatibility

Today many sdks support elicitation via an in-line but async fashion which waits for the elicitation response before sending the tool call response on the original SSE stream, this works for MCP Servers that are a single-process or can ensure sticky routing of requests.

```
def my_tool():
  do_work()
  await elicit_more_info()
  do_more_work()
  return tool_result
```

SDKs MAY continue to support this style of elicitation for existing tools and for backwards compatibility, however, they SHOULD mark this pattern as legacy/deprecated.

Moving forward examples and SDKs need to support the new style of elicitation where the code can not assume the same process is handling both tool calls. This programming model is less appealing, however it ensures that MCP Servers can go from a single process Stdio MCP server to a multi-process remote MCP Server without major rewrites, and ensures we have a single recommended way to do elicitation moving forward.

```
def my_tool(request):
  if(request.requestState):
      state = decode(request.requestState)
  if(request.inputResponses):
      additionalInfo = decode(request.inputResponses)

  do_work(state, additionalInfo)
  if(more_info_needed):
    return IncompleteResponse();
  else
    do_more_work()
    return tool_result
```

Other options considered here were to have two separate programming models that developers could choose between based on their MCP Server deployment single-process or multi-process to continue to support the await semantics, however this would have added complexity to the developer experience, and would have made it more difficult for developers to switch between single-process and multi-process deployments.

Security Implications

Because `requestState` passes through the client, malicious or compromised clients could attempt to modify it to alter server behavior, bypass authorization checks, or corrupt server logic. To mitigate this, we require servers to validate this state as described in the protocol requirements above.

Reference Implementation

TBD

Acknowledgments

Thanks to Luca Chang (@LucaButBoring) for his valuable input on how to integrate input requests into Tasks.

SEP-2468 Recommend Issuer (iss) Parameter in MCP Auth Responses

Final · Standards Track · Created 2026-03-25

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-03-25
- **Author(s)**: Emily Lauber (@EmLauber)
- **Sponsor**: @pcarleton
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2468

Abstract

This SEP proposes recommending the inclusion and requiring the validation of an explicit issuer (iss) parameter in Model Context Protocol (MCP) authorization responses to mitigate authorization mix-up attacks. By binding authorization responses to a specific authorization server identity, MCP clients can reliably detect and reject responses originating from an unexpected issuer, improving protocol robustness in multi-identity provider (IdP) environments. This SEP follows the specifications defined in RFC9207.

Motivation

The Model Context Protocol increasingly operates in environments where multiple authorization servers, identity providers, and intermediaries coexist. In such environments, OAuth mix-up attacks become a realistic threat. Mix-up attacks are when an attacker causes a client to associate an authorization response with the wrong authorization server, potentially leading to token leakage or privilege escalation.

OAuth specifications describe two mitigations for mix-up attacks: requiring issuer (*iss*) parameter or using a unique redirect_uri for each issuer a client interacts with. A unique redirect_uri per issuer is not possible with Client ID Metadata Documents (the recommended registration approach) and is operationally expensive with Dynamic Client Registration. As such, the recommendation is for MCP environments to leverage the issuer mitigation.

Requiring an explicit iss parameter in MCP authorization responses provides a simple, interoperable, and well-understood mechanism to bind responses to the correct authorization server and prevent mix-up attacks by construction. Since not every authorization server sends the issuer parameter though, this SEP proposes a MUST for clients to validate issuer if provided and a SHOULD for authorization servers supporting MCP scenarios. Future SEPs and releases may change the SHOULD to a MUST.

Specification

Issuer Parameter Requirement

MCP authorization servers SHOULD include an issuer (*iss*) parameter in authorization responses, including error responses, as defined in RFC9207. Authorization servers that do so MUST advertise it by setting `authorization_response_iss_parameter_supported: true` in their authorization server metadata.

The `iss` parameter MUST:

- Exactly match the issuer identifier advertised via metadata discovery
- Be a URL that uses the `https` scheme without query or fragment components (RFC 8414 Section 2)

Client Validation Requirements

MCP clients MUST validate the *iss* parameter in authorization responses by:

- Determining the expected issuer for the authorization request
- Comparing the received *iss* value against the expected issuer
- Rejecting the authorization response if the values do not match exactly

If issuer validation fails, the client **MUST** treat the response as invalid and abort the authorization flow.

Rationale

The iss value is already used in OpenID Connect and JWT-based token validation. Extending its use to MCP authorization responses:

- Leverages existing ecosystem knowledge and tooling
- Avoids introducing MCP-specific security mechanisms
- Provides a clear and auditable security for deployments

Alternatives considered

Introducing MCP-specific issuer binding fields

- Rejected in favor of reusing established OAuth/OIDC mechanisms.

Requiring unique redirect_uri per issuer

- CIMD metadata documents are static and cannot enumerate every issuer; with DCR it is technically possible but DCR has operational drawbacks in MCP deployments that make it undesirable to depend on for a security property. RFC 9207 works uniformly across registration approaches.

Discarding `iss` when the server does not advertise support (strict RFC 9207 §2.4 SHOULD)

- RFC 9207 §2.4 recommends that clients SHOULD discard responses carrying `iss` from servers that do not set `authorization_response_iss_parameter_supported`, but explicitly leaves the decision to local policy ("specific guidance is out of scope"). This SEP specifies comparison instead. The recorded issuer always comes from a metadata document the client has already validated per RFC 8414 §3.3, so a present `iss` can be checked against an authentic baseline; rejection on mismatch remains unconditional, so the only behavioral difference is accepting a response whose `iss` matches that baseline — which is not a relaxation. In practice, authorization servers often begin emitting `iss` before their metadata is updated, and discarding in that window would reject legitimate flows without security benefit.

Backward Compatibility

The `iss` parameter is additive on the wire. Client validation introduces a behavioral change for hosts whose authorization server advertises `authorization_response_iss_parameter_supported: true` but whose callback handling does not yet pass `iss` to the SDK; those flows will be rejected until the host extracts `iss` from the redirect URI alongside `code`. SDKs are expected to widen callback signatures additively (e.g., an optional `iss` argument) so existing call sites continue to compile. Authorization servers that do not advertise support are

unaffected. The accompanying RFC 8414 Section 3.3 metadata-validation requirement restates an existing RFC MUST; clients that were not already enforcing it may surface latent issuer misconfigurations on upgrade.

Security Implications

This proposal is a mitigation against mix-up attacks; the security considerations for the mechanism itself are documented in RFC9207 Section 4. In particular, the mitigation depends on clients establishing the expected issuer before redirecting and on the comparison being an exact simple string comparison. See also the MCP security best practices.

Reference Implementation

- Go SDK: modelcontextprotocol/go-sdk#859
- TypeScript SDK: modelcontextprotocol/typescript-sdk#1957

Both record the expected issuer before redirect and compare any received `iss`, rejecting on absence only when the server advertises support.

Acknowledgments

Thanks to Sam Morrow, Max Gerber, Aaron Parecki, Stephen Halter, Nate Barbettini, Karl McGuinness, and Den Delimarsky for reviews and discussion in the Auth Mix-Up Attack Prevention working group.

SEP-2484 Require Conformance Tests for Standards Track SEPs to Reach Final Status

Final · Process · Created 2026-03-27

- **Status**: Final
- **Type**: Process
- **Created**: 2026-03-27
- **Author(s)**: Paul Carleton (@pcarleton)
- **Sponsor**: None
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2484
- **Supersedes**: SEP-1627 (Conformance Testing)

Abstract

This SEP adds a conformance test requirement to the `Accepted → Final` transition for Standards Track SEPs. Before a Standards Track SEP that changes observable protocol behavior can be marked `Final`, a conformance scenario covering its normative requirements must be merged into the conformance repository, accompanied by a structured traceability file mapping each MUST/MUST NOT and SHOULD/SHOULD NOT to a check or a documented exclusion. This keeps the conformance suite synchronized with the specification as it evolves, gives SDK maintainers an executable target for implementation, and makes SEP-1730's tier percentages a meaningful measure of spec coverage. Process and Informational SEPs are exempt, as are Standards Track SEPs with no observable protocol behavior.

Motivation

The gap between specification and implementation

The MCP specification is written in English. SDK maintainers translate that English into code, and every translation is an opportunity for drift. SEP-1730 (SDK Tiering) already depends on conformance tests (Tier 1 requires 100% pass rate, Tier 2 requires 80%), but there is no mechanism keeping the suite synchronized with the spec. When a SEP reaches `Final`, SDK maintainers implement from prose and hope they interpreted it the same way every other SDK did. Conformance tests arrive later, if at all, and when they do they sometimes reveal that two "compliant" SDKs disagree.

Why the existing reference implementation requirement is insufficient

A **reference implementation** proves the feature can be built: one valid interpretation. A **conformance test** defines what every implementation must do: the normative requirements as executable assertions. A TypeScript reference implementation tells a Rust maintainer little about whether their code is correct. A conformance test tells them precisely, and when it doesn't, the disagreement surfaces an ambiguity in the spec itself.

Keeping the conformance suite alive

The conformance suite is the yardstick SEP-1730's tier percentages measure against. If it falls behind, an SDK could be "100% compliant" while missing major specification features. Tying tests to the SEP lifecycle creates a forcing function: the suite grows exactly as fast as the spec does.

Specification

Scope

This requirement applies **only** to Standards Track SEPs that introduce or modify **observable protocol behavior**: behavior a conformant peer can detect by inspecting messages on the wire, transport-observable side effects (HTTP status codes, headers, connection lifecycle, OAuth redirects), or process-observable side effects for local transports (stdio stream content, exit codes).

The following are **exempt**:

- **Process SEPs** (governance, workflow, community structure)
- **Informational SEPs** (guidelines, best practices without normative force)
- **Standards Track SEPs with no observable protocol behavior**, for example:
  - Documentation-only clarifications of existing behavior
  - Schema annotations that do not change validation or runtime behavior
  - Security recommendations describing implementation hardening rather than wire-level requirements

The conformance suite itself is not restricted to official SDKs. Any implementation (official SDK, community SDK, or custom deployment) may run it and report a compliance percentage.

The requirement

For a Standards Track SEP in scope to transition from `Accepted` to `Final`:

1. **A conformance scenario** tagged with the SEP number is merged into the conformance repository, targeting the conformance repository's draft spec-version tag for the upcoming release.
2. **A traceability file** accompanies the scenario. See below.
3. **The scenario passes** against the SEP's reference implementation.

When that spec version is released, the scenario's spec-version tag is updated from the draft tag to the dated version as part of the normal release process. The conformance harness and the SDK under test must both recognize the draft tag as a negotiable protocol version so that the new requirements are actually exercised.

Traceability file

The traceability file is a structured file (`sep-NNNN.yaml`) in the conformance repository. It maps each normative requirement in the SEP's Specification section to the check that exercises it, or documents why it is excluded:

```yaml
sep: 1234
spec_url: https://openmodelcontextprotocol.org/specification/draft/section#anchor
requirements:
  - check: sep-1234-foo-present
    text: "MUST include `foo` in the response"
  - check: sep-1234-bar-absent
    text: "MUST NOT send `bar` before initialization"
  - check: sep-1234-qux-present
    text: "SHOULD include `qux` when available"
  - check: sep-1234-baz-rejected
    text: "MUST reject requests with invalid `baz`"

  - text: "MUST retry on 503"
    excluded: "Requires fault injection; not currently supported by framework"
    issue: https://github.com/modelcontextprotocol/conformance/issues/N
  - text: "MUST be rendered in a monospace font"
    excluded: "Client rendering; not observable at the protocol level"
```

Structured data lets tooling link check failures back to spec sections and lets the conformance CLI report coverage per SEP.

Exclusions come in two flavors. **Framework gaps** (the behavior is observable but the framework can't express it yet) should link a tracking `issue`. **Not protocol-observable** (the requirement governs client rendering, implementation internals, or similar) needs only the `excluded` reason. A SEP whose requirements are all the second kind is exempt and doesn't need a scenario at all.

The sponsor verifies the traceability file is complete: every MUST, MUST NOT, SHOULD, and SHOULD NOT (and RFC 2119 equivalents: SHALL, REQUIRED, RECOMMENDED) in the SEP's Specification section has a row. Checks for SHOULD-level requirements report as warnings rather than failures. MAY requirements do not need rows. The sponsor does not review test code; that is the conformance repository's normal PR review. What counts as a normative requirement is the sponsor's call.

Who writes the tests

The **sponsor** is responsible for ensuring a conformance scenario is written. Scenarios are authored in TypeScript; contributors unfamiliar with the conformance repository should start with its CONTRIBUTING guide. In practice the SEP author is often best positioned, since writing the test surfaces ambiguities in the normative language that are cheaper to fix before `Final` than after.

Specification text is authoritative

Conformance tests are derived from and **subordinate to** the specification text. Where a test and the spec disagree, the spec is authoritative and the test is a bug.

Conformance test disputes

If an implementer believes a merged conformance test contradicts the spec, they open an issue in the conformance repository citing the specific spec text. A test is considered disputed once a conformance maintainer applies the `disputed` label; disputed tests do not affect SEP-1730 tier assessments until resolved.

Most disputes resolve through normal issue triage: the test is fixed, the spec is clarified, or the dispute is closed with rationale. If the disagreement is fundamental (the disputing party and the conformance maintainers cannot agree on what the spec means), either party may escalate unilaterally to Core Maintainers for a ruling, though

joint escalation is preferred since the goal is to resolve ambiguity rather than win an argument. The same escalation path is available to a sponsor if a scenario PR is blocked on non-technical grounds.

Test stability and tiering

SEP-1730 tier assessments are run against a **pinned conformance release version**, not the tip of the conformance repository. New checks added to a SEP's scenario after the SEP is `Final` (whether additional edge cases or coverage of previously-excluded requirements) land in the conformance repository's main branch but only affect tier percentages when the next tiering assessment adopts a newer conformance release.

This means SDK maintainers have a stable target between tiering waves, and the conformance suite can evolve continuously without surprise regressions in tier status.

Sponsor responsibilities

SEP-1850 makes the sponsor responsible for tracking reference implementation progress before marking a SEP as `Final`. This SEP extends that responsibility: for Standards Track SEPs in scope, the sponsor also confirms that a conformance scenario tagged with the SEP number is merged with a complete traceability file, or that an exemption is documented in the SEP.

Relationship to SEP-1730 (SDK Tiering)

This SEP strengthens SEP-1730's foundation without changing its tier definitions or thresholds. Tier assessments use pinned conformance releases, so new checks do not retroactively affect tier status. Disputed tests do not count toward tier percentages until resolved.

Scenario contributions covering existing spec behavior (not tied to a new SEP) remain welcome and are not required to carry a traceability file.

Relationship to SEP-1627 (Conformance Testing)

This SEP **supersedes** SEP-1627 by accepting the conformance repository as the canonical home for conformance tests and formalizing its role in the SEP lifecycle. SEP-1627's golden-trace approach was not carried forward; the scenario-and-checks model trades language-neutral fixtures for runtime expressiveness. SEP-1627's protocol-debugger ideas remain valuable future work.

Rationale

Why gate `Final` rather than `Accepted`?

Gating `Accepted` would require tests before Core Maintainers have agreed the feature belongs in the spec, wasting effort on rejected SEPs.

That said, writing a conformance test *during* SEP drafting is often valuable: it forces precision in MUST/MUST NOT language and surfaces edge cases the prose glosses over. Authors are **encouraged** to draft a conformance scenario before Core Maintainer review, especially for SEPs with complex behavioral requirements. It is not required, because small SEPs may not justify the upfront effort, and a rejected SEP's test is wasted work.

Gating `Final` places the hard requirement where the reference implementation requirement already sits: the SEP has consensus, and the remaining work is implementation.

Why a traceability file?

Without a defined coverage bar, "has a conformance test" would be relitigated on every SEP: does one check suffice, or must every MUST be covered? The traceability file makes coverage auditable: every normative statement has a row, and every row is either a check or a documented exclusion. "Sufficient" becomes "the file is complete."

The file also makes gaps visible. A SEP with ten MUSTs and eight exclusions is a signal: either the SEP is genuinely hard to test (the tracking issues say why) or the test author stopped early (the sponsor should push back).

Why put the authorship obligation on the sponsor?

The sponsor already shepherds the SEP through review, tracks the reference implementation, and manages status transitions. Adding "ensure a conformance test is written" is a small marginal addition to an existing role, with a clear owner.

Alternatives considered

Require conformance tests in the SEP PR itself. Rejected: couples two independent review processes with different maintainers and CI.

Gate only "major" SEPs. Rejected: "major" is subjective. The observable-behavior scope is objective: either a conformant peer can detect the change, or it cannot.

Make conformance maintainers the sufficiency judges. Rejected: concentrates veto power in a group not elected to approve spec changes. The traceability-file model lets the sponsor verify completeness without reading test code.

Backward Compatibility

This SEP is **not retroactive**. SEPs that reached `Final` before this SEP takes effect are not required to add conformance tests, though contributions are welcome.

Security Implications

None directly. Conformance tests that exercise security-relevant behavior (auth flows, input validation, transport security) improve the ecosystem's security posture by catching regressions, but this SEP does not mandate security-specific coverage beyond what the underlying SEP's MUSTs require.

Reference Implementation

The conformance repository already demonstrates the scenario-tagging pattern this SEP formalizes:

- `JsonSchema2020_12Scenario` — SEP-1613
- `ElicitationDefaultsScenario` — SEP-1034
- `ServerSSEPollingScenario` — SEP-1699
- `ElicitationEnumsScenario` — SEP-1330

The structured traceability file format and the scenario scaffolding tool (`npx @modelcontextprotocol/conformance new-scenario --sep <number>`) will be added to the conformance repository before this SEP reaches `Final` .

The process change is implemented by updating `docs/community/sep-guidelines.mdx` to add the conformance check to the `Accepted → Final` transition (see the accompanying changes in this PR).

Prerequisites for Final status

Before this SEP itself can be marked `Final`, the following conformance-repository work must be complete:

- Structured traceability file format (`sep-NNNN.yaml`) and schema
- Scenario scaffolding tool
- Conformance harness supports a draft spec-version tag as a negotiable protocol version
- `MAINTAINERS.md` published and the repository listed in MCP governance documentation

These are this SEP's own reference implementation checklist, not ongoing process requirements.

SEP-2549 TTL for List Results

Final · Standards Track · Created 2026-04-09

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-04-09
- **Author(s)**: Caitie McCaffrey (@CaitieM20)
- **Sponsor**: @CaitieM20
- **PR**: https://github.com/modelcontextprotocol/specification/pull/2549

Abstract

This SEP proposes adding fields to support caching result objects returned by `tools/list`, `prompts/list`, `resources/list`, `resources/read`, and `resources/templates/list`. Two fields will be added `ttlMs` and `cacheScope`. The TTL tells clients how long the response may be considered fresh before re-fetching. This allows clients to cache feature lists and reduce reliance on server-push notifications while remaining fully backward compatible. The `cacheScope` field controls who may cache a response. TTL supplements rather than replaces the existing notification mechanism — both can coexist.

Motivation

Today, MCP clients discover server features by invoking methods on the server. These calls return the current set of features. To learn about changes, clients rely on push notifications from the server. The below table maps the Server Method to Notification Type.

Server Methods	Notification Type
`tools/list`	`notifications/tools/list_changed`
`prompts/list`	`notifications/prompts/list_changed`
`resources/list`	`notifications/resources/list_changed`
`resources/templates/list`	`notifications/resources/list_changed`
`resources/read`	`notifications/resources/updated`

This approach has several limitations:

1. **HTTP-based transports require SSE Streams**: Many clients and servers have challenges supporting long lived SSE streams which are necessary for notifications. The goal is to make SSE streams an optional optimization, but support protocol functionality without them. A TTL allows clients to poll on a predictable schedule without relying on server-push notifications.
2. **Implementation complexity**: Both clients and servers must implement notification subscription and delivery infrastructure. Many simple servers have feature lists that change infrequently (or never), yet must still support the notification machinery if they want clients to stay current.

3. **No freshness signal**: Even clients that can receive notifications have no indication of how "stable" a list is. A server whose tool list changes once a day and one whose list changes every second look identical to the client — both simply send notifications when changes occur. A TTL provides an explicit freshness hint.
4. **Alignment with web standards**: HTTP caching (`Cache-Control: max-age`) and DNS TTLs have long demonstrated that time-based freshness hints are a simple, well-understood mechanism for reducing unnecessary refetches. MCP can benefit from the same pattern.

Adding a TTL field to list responses solves all of these problems with a minimal, backward-compatible protocol change.

Specification

New interface: `CacheableResult`

A new `CacheableResult` interface is introduced as a standalone type extending `Result`. It owns the `ttlMs` and `cacheScope` fields.

Schema change (TypeScript)

```typescript
/**
 * A result that supports a time-to-live (TTL) hint for client-side caching.
 *
 * @internal
 */
export interface CacheableResult extends Result {
  /**
   * A hint from the server indicating how long (in milliseconds) the
   * client MAY cache this response before re-fetching. Semantics are
   * analogous to HTTP Cache-Control max-age.
   *
   * - If 0, The response SHOULD be considered immediately stale, The client
   *   MAY re-fetch every time the result is needed.
   * - If positive, the client SHOULD consider the result fresh for this many
   *   milliseconds after receiving the response.
   */
  ttlMs: number & { readonly minimum: 0 };

  /**
   * Indicates the intended scope of the cached response, analogous to HTTP
   * Cache-Control: public vs Cache-Control: private.
   *
   * - "public": Any client or intermediary (e.g., shared gateway, proxy)
   *   MAY cache the response and serve it to any user.
   * - "private": Only the requesting user's client MAY cache the response.
   *   Shared caches (e.g., multi-tenant gateways) MUST NOT serve a cached
   *   copy to a different user.
   *
   * Defaults to "public" if absent.
   */
  cacheScope: "public" | "private";
}
```

Semantics

A TTL is a freshness estimate, not a guarantee. Servers MAY change the underlying list before the TTL expires; servers that do so and have advertised listChanged SHOULD send the corresponding notification.

Servers MUST provide a `ttlMs` on `Results` returned by `tools/list`, `prompts/list`, `resources/list`, `resources/read`, and `resources/templates/list`.

`ttlMs` MUST be >= 0. If a server returns a negative value, clients SHOULD ignore it and treat it as 0 (immediately stale).

Condition	Client behavior
`ttlMs` = 0	The response SHOULD be considered immediately stale, The Client MAY re-fetch every time the result is needed.
`ttlMs` > 0	Client SHOULD consider the response fresh for `ttlMs` milliseconds from receipt.
Relevant notification received while TTL is active	The notification invalidates the cached response. Client SHOULD re-fetch regardless of remaining TTL.
`cacheScope` = `"public"`	Any client or shared intermediary (gateway, proxy) MAY cache and serve the response to any user.
`cacheScope` = `"private"`	Only the requesting user's client MAY cache. Shared caches MUST NOT serve a cached copy to a different user.

Freshness calculation

A client records the local time at which the response was received (`t_received`). The response is considered **fresh** while `now < t_received + ttlMs`. Once the TTL expires the response is **stale** and the client SHOULD re-fetch on next access.

Clients SHOULD NOT treat TTL as a polling interval that triggers automatic background refetches. The TTL is a **freshness hint**: the client checks freshness when it needs the list, and re-fetches only if stale. Implementations that do choose to poll SHOULD apply jitter and backoff.

Clients MAY re-fetch if they have reason to believe the data has changed, even if the TTL has not yet expired. Examples include receiving an unexpected error on a tool call indicating that the the method was not found or the parameters were invalid.

Clients MAY serve stale responses if errors occur in re-fetching results(e.g., network issues, server downtime). The TTL is a hint for how long the client can safely rely on the data, but real-world conditions may require flexibility.

Cache scope

The `cacheScope` field controls who may cache a response:

- **`"public"`**: The response does not contain user-specific data. Any client, shared gateway, or caching proxy MAY store and serve the cached response to any user. This is appropriate for lists of tools, prompts, and resource templates that are identical for all users.
- **`"private"`**: The response contains user-specific data. Only the requesting user's client MAY cache it. Shared caches (e.g., multi-tenant API gateways) MUST NOT serve a `"private"` cached response to a

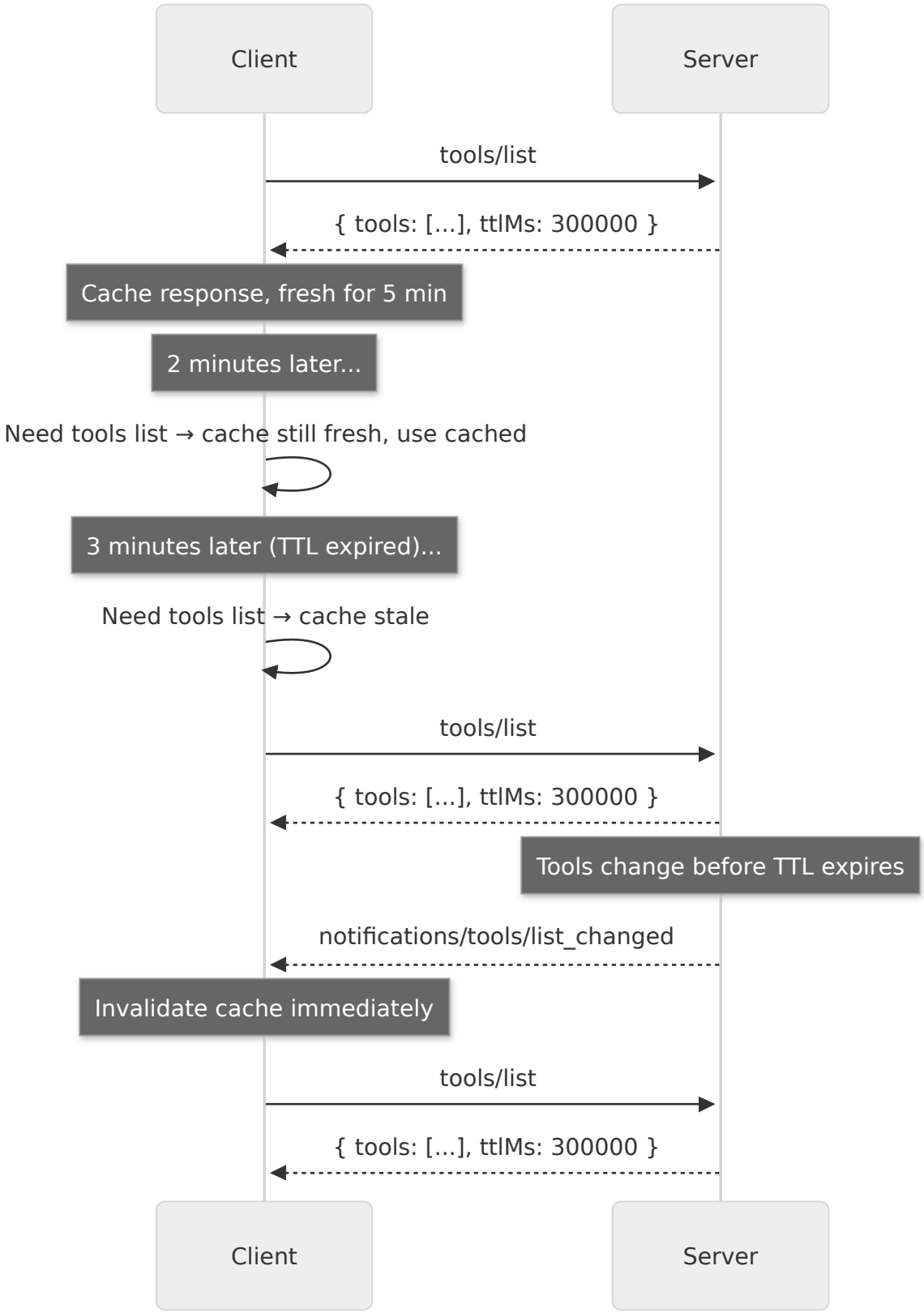
different user. This is appropriate for `resources/read` results that depend on the authenticated user, or for filtered list results that vary per user.

This design mirrors HTTP `Cache-Control: public` vs `Cache-Control: private`, applying the same well-understood semantics at the MCP protocol level.

Interaction with notifications

TTL and server-push notifications are complementary:

- A server MAY provide `ttlMs` without advertising `listChanged: true` in its capabilities. In this case the client relies entirely on TTL.
- A server MAY advertise `listChanged: true` **and** provide `ttlMs`. In this case the client can use the TTL to avoid unnecessary refetches between notifications, and the notification acts as an immediate invalidation signal.



Interaction with pagination

When a list result is paginated (includes `nextCursor`), each page is an independently cacheable response — consistent with how HTTP `Cache-Control` treats paginated resources. Specifically:

- Each page response carries its own `ttlMs` value. The freshness clock for each page starts at the time that page was received.
- Servers MAY return different `ttlMs` values on different pages (e.g., a longer TTL for early pages of a stable list, a shorter TTL for the final page).
- There is no cross-page consistency guarantee. If the underlying data changes between page fetches, clients may observe duplicates or gaps — the same trade-off that applies to HTTP paginated APIs.
- Clients that require a consistent snapshot of the full list SHOULD re-fetch from the beginning (without a cursor).
- If a cursor becomes invalid (e.g., the server returns an error for a previously valid cursor), the client SHOULD discard all cached pages and re-fetch from the beginning.

Servers MUST apply the same cacheScope to all response pages for a given list request. For example, if the first page of a `tools/list` response has `cacheScope: "private"`, all subsequent pages for that request MUST also be treated as `"private"`.

Error handling

- For backwards compatibility, If `ttlMs` is missing, clients SHOULD assume a default `ttlMs` of `0` (immediately stale) and rely on their own caching heuristics or notifications.
- If `ttlMs` is present but is a negative integer, the client SHOULD ignore it and behave as if it were 0 (immediately stale).

Rationale

Why not replace `list_changed` notifications?

Notifications provide immediate invalidation which is valuable for long-lived connections. TTL provides a complementary mechanism optimized for stateless transports and for reducing unnecessary polling. Both mechanisms serve different use cases and coexist naturally.

Why integer milliseconds for TTL?

We chose integer milliseconds over seconds as we want one unit for ttl across the MCP protocol. Tasks has uses cases for sub-second TTLs, and using milliseconds allows for a consistent representation across all TTLs in MCP.

Many existing systems use integer seconds for TTLs, but some (e.g., gRPC retry pushback) use milliseconds. The key is to choose a single, consistent unit for all TTLs in MCP. Integer milliseconds provides the necessary precision while remaining simple to implement and understand.

System	Mechanism	Notes
HTTP `Cache-Control: max-age`	Integer seconds	The most widely deployed freshness hint in web infrastructure
DNS TTL	Integer seconds	Controls how long resolvers cache DNS records
GraphQL `@cacheControl`	`maxAge` integer secs	Per-field cache hints in GraphQL responses
gRPC `grpc-retry-pushback-ms`	Milliseconds	Server-provided retry hint (different use case, similar pattern)

Why not use HTTP caching directly?

MCP is transport-agnostic. While HTTP-based transports could theoretically use `Cache-Control` headers, MCP also operates over stdio, and supports pluggable transports where HTTP headers may not be available. Embedding the TTL in the JSON response body ensures it works uniformly across all transports.

Backward Compatibility

- Existing servers that do not provide it continue to work unchanged. If a `ttlMs` field is missing, clients SHOULD assume a default ttlMs of 0 (immediately stale) and rely on their own caching heuristics or notifications, which is the current behavior.
- Existing clients that do not understand the field will ignore it, as MCP result objects permit additional properties via `[key: string]: unknown` on the `Result` base type.
- `cacheScope` is required because there is no safe default for older servers. The server must explicitly declare the intended cache scope to prevent unintended caching of user-specific data.
- No existing fields or behaviors are modified or removed.
- No capability negotiation is required.
- SDK Maintainers can choose to add defaults for ttl and cacheScope in their SDKs to simplify adoption, but this is not required for compliance.

Reference Implementation

No reference implementation yet.

Security Implications

A misconfigured or malicious server could set an excessively long TTL, causing clients to cache stale data for longer than desired. However, since the TTL is a hint and clients can choose to ignore it or re-fetch if they suspect changes, the security risk is minimal. Clients should be designed to handle unexpected TTL values gracefully.

SEP-2567 Sessionless MCP via Explicit State Handles

Final · Standards Track · Created 2026-03-11

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-03-11
- **Author(s)**: Peter Alexander (@pja-ant)
- **Sponsor**: Peter Alexander (@pja-ant)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2567
- **Related**: SEP-2575 (Make MCP Stateless), SEP-2322 (Multi Round-Trip Requests), SEP-2549 (TTL for List Results)

Abstract

This proposal removes the protocol-level session concept from MCP, replacing implicit session-scoped state with explicit, server-minted state handles that the model carries and threads through subsequent calls. SEP-2575 removes the `initialize` handshake and carries protocol version and capabilities per-request; this proposal is the complementary change that removes sessions and the `Mcp-Session-Id` header. Together they make MCP stateless at the protocol layer.

After more than a year in the spec, sessions have not converged on a consistent meaning across clients: some scope them per tool call, some per application launch, some per page load, and almost none resume them. A server author cannot predict what scope or lifetime a session will have when their server is connected to an arbitrary client, which has made the session unreliable as a container for application state. This proposal holds that application state can be served by explicit identifiers, and that the session abstraction adds constraints (fixed cardinality, undefined lifetime, uncacheable list endpoints across session boundaries) without corresponding benefit.

Under this proposal, a server that currently scopes a shopping cart (for example) to the session instead exposes a tool `create_basket()` that returns a `basket_id` and threads that ID through subsequent tool calls, e.g. `add_item(basket_id, ...)`. The model decides what is shared and what is isolated; list endpoints become cacheable across what used to be session boundaries; and agent orchestrators can freely share or not share application state as needed. Explicit state handles are not a new protocol construct — there is no schema or wire format for them. They are a tool-design pattern; the protocol change is the removal of sessions, which leaves handles as the way to express cross-call state.

Motivation

What sessions scope today

The current spec is imprecise about which behaviors are session-bound, but in practice five categories of things attach to a session's lifetime:

1. **Negotiated capabilities and protocol version.** The result of `initialize` — which protocol version is in use and which optional capabilities each side supports — is established once per session and assumed for

its duration. SEP-2575 resolves this by removing `initialize` and carrying version/capability information per-request, so this proposal treats it as already addressed.

2. **Elicitation and sampling intermediate state.** When a tool call triggers an `elicitation/create` or `sampling/createMessage` round-trip, the server has to correlate the eventual response with the original in-flight tool call — state that today lives implicitly in the session. SEP-2322 (Multi Round-Trip Requests) resolves this by carrying the correlation state explicitly through the request/response cycle, so this proposal treats it as already addressed.

3. **Application state.** The canonical example is a shopping cart: `add_item()`, `add_item()`, `checkout()`, with the cart existing implicitly per-session. This generalizes to any stateful workflow — a Playwright browser instance, a database transaction, an open file descriptor.

4. **Mutable list endpoints.** `tools/list` (and `resources/list`, `prompts/list`) can legally return different results over a session's lifetime. For example, a database server could expose a `connect_database` tool that, once called, makes `query` and `list_tables` appear in subsequent `tools/list` results.

5. **Resource subscriptions.** Subscription lifetime is tied to session lifetime. (SEP-2575 introduces `messages/listen` as the delivery channel for server-to-client notifications; subscription lifetime under that model is not re-examined here.)

With (1), (2), and (5) handled by other SEPs, this proposal addresses (3) and (4).

Problems with session scoping

The issues below apply whether sessions are mandatory (the current spec) or made optional.

Session lifetime is undefined, and servers can't design around it

The spec does not say when a session begins or ends, because it depends on the host application. In practice, deployed clients vary widely and few scope sessions to a conversation: ChatGPT creates a fresh session for every individual tool call, and Claude.ai did the same until recently;[1] most desktop and IDE clients create one at application launch and keep it for the process lifetime; web clients typically create one per page load. Almost no clients resume a prior session after a disconnect or restart, and on the server side the reference TypeScript SDK provides no public API for reconstructing a session on a different node, so multi-node deployments cannot honor resumption even when a client attempts it.[2] A subagent might share its parent's session or get its own — there is no convention.

This matters because server authors are the ones deciding what to scope to the session, and they need to know what a session corresponds to in order to do that correctly. A Playwright server that ties a browser instance to the session needs to know whether that means one user turn, one agent process, or one long-lived chat. The spec does not specify this and different hosts give different answers, so the server is designing against an abstraction whose semantics it does not control.

The practical consequence is that session-scoped application state often does not survive. Against a per-tool-call client it is destroyed before the next call; against a per-app-launch client it is shared across every conversation in the window and then lost on restart; against any client that does not resume, it is gone when the app restarts. Servers that appear to be using session state successfully are usually stdio servers relying on process lifetime, which is a property of the transport rather than the protocol.

List endpoints cannot be cached across sessions

Because `tools/list` may be session-dependent, a client cannot assume a result fetched in one session is valid in the next. Every new session must re-fetch, even when the server's tool set is fixed at build time and never changes, which is the common case.

The Python SDK's own design issue for client-side list caching lists "what is the cache key — per-session or per-server-URL?" as an open question,[3] and gateway implementers have shipped per-session caching specifically because they could not assume cross-session validity from the spec.[4] Every list endpoint must be treated as potentially session-scoped, so each must be re-fetched per session to be safe.

For hosts that regularly spawn subagents, this is a multiplier on the hot path. The possibility that a server is session-scoped forces `O(subagents × servers)` calls to `tools/list`: every subagent, for every server, every time, even if the underlying tool set has not changed since the orchestrator first connected. The client cannot skip the call because it cannot know in advance which servers are session-scoped. For an orchestrator spawning many short-lived subagents, this overhead can exceed the protocol traffic of the actual tool calls. Under this proposal the same workload is `O(servers)`: the orchestrator fetches each list once and every subagent reuses the cached result.

If list endpoints were a function only of the server deployment and the authenticated principal, clients could cache them and invalidate on an explicit signal. SEP-2549 specifies such a signal (a server-advertised TTL plus `notifications/*/list_changed`), but its caching model is only sound if the list does not also vary per session. Removing sessions makes that model safe; a subagent can then inherit its parent's cached lists at zero cost.

Cardinality is fixed at one per session

Session state has a cardinality of exactly one per session. The model gets one cart, one browser, one of whatever the server scopes to the session; it cannot have two, and it cannot have zero.

This is a problem when different pieces of state need different scopes. Consider an orchestrator that spawns several subagents to independently research products to buy. The subagents should add to the same shopping cart (they are collaborating on one order) but each needs its own browser state (they are browsing different sites in parallel).

No session boundary satisfies both:

Session model	Cart (want: shared)	Browser (want: isolated)
Subagents share parent's	✓ shared	✗ shared (clobbers)
Subagents get their own	✗ isolated	✓ isolated

With explicit IDs the orchestrator calls `create_basket()` once, passes the resulting `basket_id` to each subagent, and each subagent separately calls `create_browser()` for its own `browser_id`. The model decides what is shared and what is isolated per piece of state, rather than having one scope imposed on everything.

The same lack of an identifier also means session state is not addressable from outside the session that created it. A cart created in one chat is invisible to another chat; if a user wants to resume work in a new conversation, hand something off to a different agent, or share state with a colleague, the session model provides nothing to refer to it by. An explicit `basket_id` can be passed to any of those.

Specification

Summary of changes

1. **Remove the session concept from the protocol.** The `Mcp-Session-Id` header is removed and the spec language describing session lifecycle and session-scoped behavior is deleted. The protocol is sessionless at every layer. (SEP-2575 removes the `initialize` handshake but explicitly defers session removal to this proposal.)

2. **List endpoints are session-independent.** With no session, the results of `tools/list`, `resources/list`, and `prompts/list` have no per-session or per-connection scope to depend on. Lists can still change for other reasons (server deployment, auth changes); caching and invalidation mechanics for those are specified separately in SEP-2549.

3. **Stateful workflows use explicit handles.** With sessions gone, servers that need to maintain state across tool calls do so by returning an identifier from a creation tool and accepting it as a parameter on subsequent calls.

That third point is **not a protocol change**. There is no `handles/*` method, no handle type in the schema, no wire-level concept of a handle at all. From the protocol's perspective a handle is a string in a tool result and a string in a tool argument, indistinguishable from any other tool data. "Explicit state handles" is a tool-design pattern that the spec documents and recommends — in the same way it might document pagination or error-message conventions — not something it implements. The normative content of this SEP is the removal in (1); (2) follows from it, and (3) is the guidance that fills the gap.

Explicit state handles

Pattern

Where a server would previously have relied on implicit session-scoped state — `add_item` calls operating on a per-session cart — it instead exposes an explicit creation tool that returns a handle:

```
// → tools/call
{ "name": "create_basket", "arguments": {} }

// ← result
{ "content": [{ "type": "text", "text": "Created basket bsk_a1b2c3" }],
  "structuredContent": { "basket_id": "bsk_a1b2c3" } }
```

The model then threads that handle through subsequent calls as an ordinary argument:

```
// → tools/call
{ "name": "add_item",
  "arguments": { "basket_id": "bsk_a1b2c3", "sku": "shoes" } }

// ← result
{ "content": [{ "type": "text", "text": "Added shoes to bsk_a1b2c3 (1 item)" }] }

// → tools/call
{ "name": "checkout",
  "arguments": { "basket_id": "bsk_a1b2c3" } }
```

Nothing here is a protocol extension: `basket_id` is an ordinary string field in `structuredContent` and an ordinary string argument to subsequent tools. This pattern is already the norm in widely-deployed remote MCP servers that manage durable resources:

Server (official, remote)	Create tool → returned ID	Operate tools taking that ID
Linear	`create_issue` → issue id	`get_issue`, `update_issue`, `create_comment`
Notion	`notion-create-pages` → page id	`notion-update-page`, `notion-move-pages`
GitHub	`create_pull_request` → PR number	`pull_request_read`, `update_pull_request`, `merge_pull_request`
Stripe	`create_customer` → customer id	`create_invoice`, `list_subscriptions`

The approach can be adopted for less-persistent objects (a browser context, an in-progress cart) by giving the created object a limited lifetime, and/or limiting its discoverability to the principal that created it. The server owns the state, the client holds a name for it, and authorization is checked on every call.

Guidance for servers

None of the following is normative. Handles are a tool-design pattern, not a protocol feature, and servers are free to shape them however fits their domain. The pattern works best when:

- **Handles are opaque.** A handle that encodes internal structure (`cart_user42_2026-03-11`) invites clients to parse it or models to guess it; an opaque handle such as `bsk_a1b2c3` does not.
- **Possession is not authorization (where auth exists).** For authenticated servers, validate `(handle, auth_context)` on every call; handles will end up in chat logs, copy-paste buffers, and subagent prompts. For unauthenticated servers, where the handle is necessarily a bearer token, generate it with at least 128 bits of cryptographically secure entropy and bound its lifetime. See Security Implications.
- **Durability is documented in the tool description.** Handles outlive connections by design, so "the state lasts until the connection closes" is no longer applicable. Put the policy in the `create_*` tool's description — "returns a basket_id; baskets expire after 24h idle" — so it is visible to the model when it decides to create state. A policy only in server documentation is not visible to the model.
- **Expired handles return useful errors.** When a tool receives a handle for state that has expired or been destroyed, the error should say so — "basket `bsk_a1b2c3` has expired" rather than "invalid argument". A clear expiry error lets the model recover by calling `create_*` again; an opaque error typically leads to retries or failure.
- **Creation takes parameters.** `create_context(cluster="staging")` is preferable to `create_context()` followed by `set_cluster(ctx, "staging")`: one round-trip instead of two, and the state cannot exist half-configured.
- **Cleanup is available.** A `destroy_*(handle)` tool lets models release resources. A `list_*()` tool lets a model recover after losing track of what it created. Neither is required.

Guidance for clients

From the client's perspective, a handle is an ordinary string in a tool result. The main client responsibility is ensuring that string survives context compaction; if the conversation is summarized and the handle is in the discarded portion, the state is orphaned. Clients that track tool-call results across compaction boundaries handle this already.

Session-independent list endpoints

With sessions removed, list endpoints no longer have a session to vary against. This is the only constraint this SEP places on `tools/list`, `resources/list`, and `prompts/list`: there is no longer a per-session or per-connection

scope for their results to depend on. This does not preclude varying the list by the authorization presented on the request: credentials are carried on each request, so a server returning different tool sets to different principals or scopes is relying on per-request input, not connection state. Lists can also still change over time for other reasons — a server deploys a new version, a user's plan or granted scopes change — and this SEP does not enumerate or restrict those.

How clients learn that a cached list has gone stale is the subject of SEP-2549, which defines a server-advertised TTL on list responses and the interaction with `notifications/*/list_changed`. The two SEPs are complementary: this one removes the session as a source of variation, so there is a stable thing to cache; SEP-2549 specifies how long to cache it and when to invalidate.

One consequence of the constraint above is that servers can no longer mutate list results as a side effect of other requests; the pattern from the Motivation — where calling `connect_database()` makes `query` and `list_tables` appear in subsequent `tools/list` results — is no longer permitted. For the same effect, the server exposes `query` and `list_tables` unconditionally at list time and has them take a `connection_id` argument returned by `connect_database()`. A `query` call without a valid `connection_id` fails with an error directing the model to call `connect_database()` first; the dependency is expressed in the tool's input schema and description rather than in the list result.

Consequential spec edits

Beyond removing the §Session Management section itself, several other places in the current spec define behavior in terms of session scope and need re-scoping:

- **JSON-RPC request ID uniqueness.** The spec currently requires that a request `id` "MUST NOT have been previously used by the requestor within the same session." The purpose of `id` is for the sender to correlate an incoming response with the request that produced it; the receiver only echoes it. With sessions removed, the constraint is re-scoped accordingly: a sender MUST NOT issue a request whose `id` matches that of another request it has sent and not yet received a response for. This is transport-agnostic, sufficient for correlation under every transport, and is what the TypeScript and Python SDKs already do via a monotonically increasing counter per client object. (JSON-RPC 2.0 §4 itself imposes no uniqueness requirement — it only requires the receiver to echo the `id` — so this remains an MCP-level constraint.)
- **SSE event ID uniqueness.** The spec currently scopes SSE event IDs as "globally unique across all streams within that session." With sessions removed, the constraint is simply that the ID is globally unique across all streams the server manages, so that a `Last-Event-ID` resolves to a single stream. The existing guidance that event IDs encode the originating stream already implies this.
- **Pagination cursor validity.** The spec currently advises clients not to "persist cursors across sessions." With sessions removed, this advice disappears. Cursor stability and snapshot consistency are outside the scope of this proposal.
- **List-endpoint variability.** The `tools/list`, `resources/list`, and `prompts/list` pages each say results "MAY change over the lifetime of the connection." These are re-scoped per §Session-independent list endpoints: results MAY change over time but MUST NOT vary per-connection or as a side effect of other requests on the connection.
- **Wording.** A handful of phrases that use "session" descriptively — "stateful session protocol" in the architecture overview, "available during the session" in capability negotiation, "same logical session" in authorization, the elicitation prohibition on associating state "with session IDs alone," and similar — are reworded or removed. These carry no semantic change beyond the session removal itself.

Rationale

Why remove sessions rather than just default them off?

SEP-2575 already addresses making MCP work behind load balancers and without sticky routing by removing the `initialize` handshake. The reasons for also removing sessions, rather than retaining them as an opt-in

capability, are:

- **Opt-in sessions still prevent list caching.** A client cannot cache `tools/list` across session boundaries unless it knows the server does not opt into session-scoped mutation, and it cannot know that in advance. The client therefore re-fetches per session per server even though few servers opt in. The `O(subagents × servers)` cost from the Motivation section is caused by sessions being possible, not by sessions being used, so making them optional does not remove it.
- **The primitive influences server design.** Offering session-scoped state in the spec leads server authors to use it for workflows that would be better served by explicit IDs.
- **Fewer primitives reduce implementation surface.** Every protocol concept must be implemented by SDK authors, documented, and learned by new users.

Expressiveness

A session provides exactly one scope per connection. Explicit IDs provide as many scopes as the model creates, and each can be shared or isolated independently. Anything expressible with a session is expressible with a single ID the model creates at the start of the conversation; the converse does not hold.

Resumption

Because handles appear in tool results, they are part of the chat transcript. Any client that persists its chats — which is most of them — therefore persists the handles automatically. Reopening a conversation after an app restart, a page reload, or on a different device puts the handle back in front of the model with no additional resumption machinery, and this behavior is consistent across clients. Session-based state, by contrast, requires the client to persist and resend `Mcp-Session-Id` out of band, which (as covered in the Motivation) almost no clients do.

Anticipated objections

Garbage collection

Sessions provide a lifecycle signal — when the session ends, state is freed. Without it, the model might forget to call `destroy_basket()`, and state leaks.

However, sessions do not deliver this reliably in practice. As covered in the Motivation, real clients either never end the session (per-app-launch), end it constantly (per-tool-call), or end it at moments uncorrelated with the conversation (page reload, network blip). Stateless HTTP servers behind load balancers never see a connection-close. Servers already rely on TTL-based expiry today; the session boundary is not what performs cleanup.

Explicit IDs with a documented durability policy ("baskets expire after 24h idle") is the same mechanism, made explicit.

Models have to carry the IDs forward

With implicit session state, the server tracks the identifier; with explicit IDs, the model is responsible for threading `basket_abc123` through every relevant call. The failure modes are hallucinating a slightly-wrong ID, or the ID falling out of context when the conversation is compacted.

Models already carry opaque identifiers through conversations routinely — file paths, URLs, commit hashes, PR numbers, UUIDs returned from prior tool calls — and current models do this reliably. Compaction is the harder case, but it affects any long-horizon state: if the compactor drops live tool-call results, the model loses track of what is in the session-scoped cart as well, not just the cart's ID.

IDs in chat history

A `basket_id` that can be pasted anywhere could become an unauthenticated capability in the user's chat log.

For authenticated servers, the ID should be a name, with the server checking `(id, auth_context)` on every call. Google Doc IDs sit in URLs and browser history; access is controlled by ACL, not ID secrecy. The same applies here.

For servers without authentication, the ID is necessarily a bearer token — possession is the only thing the server can check. In that case the handle should follow standard practice for unguessable capability tokens: generated from a cryptographically secure random source with at least 128 bits of entropy (e.g. UUIDv4, or 22+ characters of URL-safe base64), never derived from predictable inputs, and given a bounded lifetime. This is the same posture as other ephemeral public IDs in common use — "anyone with the link" share URLs, password-reset tokens, Stripe Checkout session IDs — and carries the same tradeoff: convenient, but anyone who obtains the token has access for its lifetime.

Breaking change

Sessions are in the spec today; removing them breaks anyone relying on them.

An automated survey of a 1000-repo random sample of open source MCP servers (classified by per-repo LLM analysis) found:

Category	Share	Migration
No application-level reference to MCP session ID	90.0%	None
`Map<sessionId, Transport>` routing (TS SDK boilerplate)	3.5%	Removed by a sessionless SDK transport
Transport setup only (`sessionIdGenerator` , never read)	2.8%	Delete one constructor option
Session-keyed application state	2.5%	Migrate to explicit handles or auth principal
Proxy / gateway sticky routing	0.7%	Needs designed replacement
Auth binding (JWT claims, PKCE verifier keyed on session)	0.5%	Replace with server-generated nonce or token subject

The bolded rows are the repos that use the session ID for application semantics. The hardest-hit category — gateways that spawn one upstream per session — needs a designed replacement rather than a mechanical edit; see Backward Compatibility.

Backward Compatibility

This is a **breaking change** for servers that rely on protocol-level session state. The migration path depends on server category:

Stdio servers using process-lifetime state. These are the most common stateful servers today. Mechanically they are not broken by this proposal in their default deployment — the process lifetime still exists, and a server that keeps a single in-memory browser instance per process continues to function with a stdio client that spawns one process. However, such servers SHOULD NOT rely on process-lifetime state and SHOULD migrate to explicit

handles. Process lifetime has the same undefined-scope problem this SEP removes for HTTP (whether the process corresponds to one conversation, one application launch, or something else is up to the host), and a server that depends on it cannot offer equivalent behavior over HTTP, where there is no process per client. Stdio servers never had `Mcp-Session-Id`, so the header removal itself does not affect them.

HTTP servers using `Mcp-Session-Id`. These are less common and must migrate to explicit handles. The migration is mechanical: replace the session-scoped state map with a handle-keyed state map, add a `create_*` tool, add the handle as a parameter to stateful tools.

Servers using session ID as a telemetry key. Some servers tag traces, logs, or rate-limit buckets with the session ID to correlate activity within a session. This already worked inconsistently across clients — against per-tool-call clients every event lands in its own bucket, and against clients that don't resume the correlation breaks at every restart. These use cases need to move to a different scoping mechanism, typically the authenticated principal (bearer token subject, API key) or a request-level correlation ID.

Proxies and gateways using session ID for sticky routing. Gateways that route by `Mcp-Session-Id` lose their routing key — but they only needed one because their upstreams were stateful. If the upstream is stateless (or migrates to explicit handles, where the state key is in the tool arguments and any replica can serve it from shared storage), the gateway needs no sticky routing at all. The residual case is gateways that bridge HTTP to stdio by spawning one subprocess per session; those need a different correlation key, which is a transport-layer concern (route by authenticated principal, or a cookie / gateway-issued header) rather than something this SEP defines.

Servers binding auth artifacts to session ID. A small number of servers store OAuth PKCE verifiers, session→user pinning maps, or JWT claims keyed on the session ID. In the PKCE case the server is already passing a correlation value through the OAuth `state` parameter (the browser callback is not an MCP request and never carried `Mcp-Session-Id`), so the change is to put a server-generated nonce in `state` instead of the session ID. Session→user pinning was a defense against the session-routing/auth decoupling described in Security Implications and is not needed once every request is independently authenticated. The migration is mostly mechanical, though worth a review since auth code is involved.

Clients. Clients become simpler: they no longer track or resend session identifiers, or need to determine whether a given server is stateful. List-endpoint caching becomes safe.

Rollout is a clean break: sessions are removed in the next spec version, with no deprecation window. Servers that currently rely on session-scoped state stay on the current protocol version until they have migrated to explicit handles. Protocol version negotiation already handles mixed-version deployments — a client that supports both versions speaks the old protocol to an unmigrated server and the new one to everyone else. This avoids shipping a version where clients support both modes simultaneously, which would prevent the caching benefit (a client cannot cache list endpoints if any connected server might be session-scoped).

Security Implications

Handle exposure

The main security consideration introduced by this SEP is that handles will end up in places session IDs did not — chat logs, subagent prompts, copy-paste buffers, potentially other users' screens.

This is a change in exposure surface, not a new class of vulnerability. Session IDs are already capability-bearing in practice: the Python SDK's stateful session manager, for example, routes by `Mcp-Session-Id` alone without verifying that the authenticated identity on the request matches the one that created the session, so a leaked session ID allows hijack by any other authenticated principal.[5] The "validate `(id, auth_context)` on every call" guidance below applies equally to today's session IDs and to explicit handles; this SEP makes the requirement more visible because handles are more visible.

For authenticated servers, the recommended posture is the same one Google Doc IDs and GitHub PR numbers take: the ID identifies the resource, and the auth context on the request determines access. Servers that validate `(handle, auth_context)` on every call are unaffected by handle exposure.

For unauthenticated servers there is no auth context to check, so the handle is a capability token. These should be generated with at least 128 bits of cryptographically secure entropy, never derived from predictable inputs, and given a bounded lifetime — the same practice as "anyone with the link" share URLs or password-reset tokens. Exposure of such a handle grants access for its lifetime; servers should size that lifetime accordingly.

This is guidance, not a protocol requirement, since the protocol has no handle concept to enforce against.

Reference Implementation

All official SDKs except PHP already provide a stateless mode, implemented as not generating a session ID (e.g. `sessionIdGenerator: undefined` in the TypeScript SDK, `stateless_http=True` in the Python SDK). This SEP makes that mode the only option for servers speaking the new protocol version. SDKs that support multiple protocol versions retain the session-ID-generating code path for older versions; the change is that it is no longer reachable when the negotiated protocol version is the one this SEP introduces.

Future Work

This SEP deliberately does not introduce a protocol-level concept of a handle: from the wire's perspective `basket_id` is an ordinary string. A consequence is that nothing marks `basket_id` as a state handle to the client or model — the relationship between `create_basket`'s output and `add_item`'s input is inferred from naming and tool descriptions, not declared.

A follow-up proposal could make that relationship explicit, for example via shared JSON Schema `$defs` referenced across a server's tool input and output schemas, or via a tool annotation that marks a result field as a handle. That would let orchestrators identify which values are live state (for compaction, hand-off, or cleanup purposes) without parsing tool descriptions. It is left out of scope here to keep this SEP to the minimum needed to remove sessions.

Footnotes

1. microsoft/playwright-mcp#1045, Sep 2025 — server author reports both ChatGPT and Claude.ai closing the session after each tool call, dropping browser state; "Connector tool calls generating fresh MCP session each invocation", OpenAI Developer Community, Nov 2025. ↩

2. modelcontextprotocol/typescript-sdk#1658, Mar 2026 — `StreamableHTTPServerTransport` stores session state in private instance fields with no API to rehydrate from external storage. ↩

3. modelcontextprotocol/python-sdk#2108, Feb 2026. ↩

4. agentgateway/agentgateway#1510, Apr 2026. ↩

5. modelcontextprotocol/python-sdk#2100. ↩

SEP-2575 Make MCP Stateless

Final · Standards Track · Created 2025-06-18

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2025-06-18
- **Author(s)**: Jonathan Hefner (@jonathanhefner), Mark Roth (@markdroth), Shaun Smith (@evalstate), Harvey Tuch (@htuch), Kurtis Van Gent (@kurtisvg)
- **Sponsor**: Kurtis Van Gent (@kurtisvg)
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2575

Abstract

A truly stateless protocol, where every request is self-contained and can be understood in isolation, is highly desirable for its inherent simplicity, scalability, and reliability. The current Model Context Protocol (MCP) is not stateless by default. The specification requires an initialization handshake that establishes a session state between the client and server, which persists for the duration of the connection.

This inherent statefulness makes it difficult to run MCP at scale. Placing an MCP server behind a standard load balancer, for example, is challenging because a client's session is coupled to the specific server instance holding its state.

This proposal outlines a series of changes to **enable stateless MCP as the default**, embracing a "pay as you go" model for protocol complexity and state. Under this model, we provide simple, stateless features by default and only introduce the overhead of stateful, long-lived connections for cases where that functionality is actually required.

Specifically, this SEP proposes removing the state-establishing initialization handshake and replacing it with discrete, stateless alternatives. This initial step allows each request to be processed independently, simplifying server-side logic and paving the way for robust, scalable deployments.

Motivation

The Model Context Protocol (MCP) specification currently mandates a stateful initialization handshake. This design choice creates significant challenges for scalability, reliability, and implementation simplicity. This SEP is motivated by the need to address these shortcomings.

The Problem with Statefulness

The core issue is that a server must retain session state from previous requests to understand subsequent ones. This is in direct opposition to the design of modern, cloud-native systems which favor stateless services for their resilience and scalability.

1. **Impediment to Scalability:** The most critical issue is the difficulty of load balancing stateful MCP. A simple stateless load balancer (e.g., L4/L7 round-robin) cannot be used, as it would route a client's requests to different backend servers, none of which would have the correct session state. Operators are forced to implement complex and fragile solutions like sticky sessions, which bind a client to a specific

server. This complicates infrastructure, can lead to uneven load distribution, and makes horizontally scaling the service non-trivial.
2. **Poor Resilience and Fault Tolerance:** In a stateful model, if the specific server instance handling a client session fails, that session state is lost. The client must detect the connection failure, re-establish a connection (likely to a new server instance via the load balancer), and perform the entire initialization handshake again. This process is disruptive and inefficient, adding complexity around "resumability".
3. **Increased Implementation Complexity:** The current model imposes a significant burden on developers.
    - **Server-side:** Developers must implement logic to create, manage, and eventually garbage-collect per-client session state. This is a common source of bugs and memory leaks.
    - **Client-side:** Developers must write complex code to manage a persistent connection and handle the inevitable network failures and reconnections, including the logic to resynchronize state after a disconnect.

Design Principles

This proposal establishes a "pay as you go" model for protocol complexity, guided by the following principles in order of preference:

1. **Prioritize Stateless-ness:** Whenever possible, a request must be self-contained, providing all information the server needs to process it without relying on state from previous requests.
2. **Prefer State References:** If a fully stateless exchange is not practical, references to state should be passed in every request.
3. **Treat Statefulness as a Last Resort:** The complexity of stateful logic and long-lived streaming connections should only be accepted when no simpler alternative exists to solve a critical use case.

Transport Consistency

It is critical that these stateless principles are applied consistently across all transports. Keeping the `stdio` and `http` implementations in sync ensures a **unified developer experience**, allowing the core protocol semantics to be learned once and applied everywhere. This consistency simplifies the creation of transport-agnostic libraries and tooling, and prevents protocol fragmentation where different transports behave in fundamentally different ways. A single, coherent protocol model is essential for a healthy ecosystem.

Specification

Overview

This specification fundamentally refactors the MCP interaction model to be **stateless-first**. Currently, MCP requires a mandatory 3-way initialization handshake before any resources can be exchanged. This handshake negotiates and establishes several key pieces of information:

1. MCP Protocol Version
2. Server Capabilities and `serverInfo`
3. Client Capabilities and `clientInfo`

The requirement of this initialization handshake **enforces the establishment of a state** that is expected to persist for subsequent communication between client and server. Furthermore, by bundling these negotiations into a single initialization phase, the specification creates an implied link between them, particularly between the exchange of capabilities and a mandatory connection lifecycle.

This proposal is to **remove the initialization handshake** and "unbundle" its functions into discrete, stateless components. We will provide new, more clearly defined mechanisms for clients and servers to exchange this information without a mandatory state-creating cycle.

> **Note:** Session management (both transport-level and application-level) is addressed separately by SEP-2322 and SEP-2567. This SEP focuses exclusively on removing the initialization handshake and providing stateless alternatives for version negotiation, discovery, and capabilities.

Protocol Version

To make requests self-contained, metadata previously negotiated during the handshake must now be included with **every request**.

HTTP

For the HTTP transport, protocol version MUST be passed as an **HTTP header**. The header value MUST match the value provided in the request payload's `_meta` field; otherwise the server MUST return a `400 Bad Request` (see SEP-2243).

- `MCP-Protocol-Version: 2025-06-18`
  - **Purpose**: To inform the server which version of the MCP specification the client is using for this specific request.
  - **Requirement**: This header is **MANDATORY**. Servers should reject requests with a missing or unsupported version.
  - This header MUST match the value provided in the Request as specified below.

Per-request Version

The `protocol-version` MUST be embedded directly within the `_meta` field of the request payload. For HTTP, this _meta MUST match the associated HTTP header, or else the server should return a 400 Bad Request.

The following diff illustrates the required changes to `RequestMetaObject`:

```
export interface RequestMetaObject extends MetaObject {
  progressToken?: ProgressToken;
+ /**
+  * The MCP Protocol Version being used for this request.
+  */
+ "io.modelcontextprotocol/protocolVersion": string;
  // Additional per-request fields (clientInfo, clientCapabilities, logLevel)
  // are introduced in the Per-Request Client Capabilities section below.
}
```

Unsupported Protocol Versions

If a server receives a request with a protocol version it does not implement (whether the version is unknown to the server or is a known version the server has chosen not to support, such as an experimental or draft version), it MUST return a JSON-RPC error response. For HTTP, the response status code MUST be `400 Bad Request`. The error MUST conform to the following structure:

```
export const UNSUPPORTED_PROTOCOL_VERSION = -32022;

export interface UnsupportedProtocolVersionError extends Omit<
  JSONRPCErrorResponse,
  "error"
> {
  error: Error & {
    code: typeof UNSUPPORTED_PROTOCOL_VERSION;
    data: {
      /**
       * An array of protocol version strings that the server supports.
       */
      supported: string[];
      /**
       * The protocol version that was requested by the client.
       */
      requested: string;
    };
  };
}
```

Version Negotiation Flow

Without an initialization handshake, version negotiation happens inline:

1. The client sends a request with its preferred protocol version in the `MCP-Protocol-Version` header and `io.modelcontextprotocol/protocolVersion` `_meta` field.
2. If the server supports that version, it processes the request normally.
3. If the server does not support the requested version, it returns an `UnsupportedProtocolVersionError` containing its list of `supported` versions.
4. The client selects a mutually supported version from the list and retries.

Alternatively, a client **MAY** call `server/discover` first to learn the server's supported versions before sending any other requests.

Discovery for Server Capabilities

To allow clients to adapt to different server implementations, this specification introduces a **discovery RPC**. This provides a standard mechanism for a server to advertise its supported protocol versions and capabilities.

Servers **MUST** implement `server/discover`. Clients **MAY** call it but are not required to — a client is free to invoke any RPC without first calling the discovery endpoint. If a client calls an unsupported RPC, the server **MUST** return a `Method not found` JSON-RPC error (`-32601`). For HTTP, the response status code MUST be `404 Not Found`.

`server/discover` RPC

- **Purpose**: To allow a client to query the server for its supported protocol versions, capabilities, and other metadata.

Request Schema:

```ts
export interface DiscoverRequest extends Request {
  method: "server/discover";
  params?: {};
}
```

Response Schema:

```ts
export interface DiscoverResult extends Result {
  /**
   * A list of MCP Protocol Version strings that this server supports.
   * The client should choose a version from this list for use in
   * subsequent requests.
   */
  supportedVersions: string[];

  /**
   * An object detailing the capabilities of the server.
   */
  capabilities: ServerCapabilities;

  /**
   * Information about the server software implementation.
   */
  serverInfo: Implementation;

  /**
   * Natural language instructions describing how to use the server and
   * its features. This can be used by clients to improve an LLM's
   * understanding of available tools (e.g., by including it in a system prompt).
   */
  instructions?: string;
}
```

Per-Request Client Capabilities

To complete the decoupling from the initial handshake, client capabilities are no longer negotiated once at initialization. Instead, a client **MUST** specify its capabilities on every request. This ensures the server is always fully informed about what optional features the client can handle for that specific transaction. An empty capabilities object means the client supports no optional capabilities — servers **MUST NOT** infer capabilities from prior requests.

Per-Request Metadata Schema

Every request's `_meta` carries a small set of fields that previously lived in the initialization handshake. The full `RequestMetaObject` shape:

```
export interface RequestMetaObject extends MetaObject {
  progressToken?: ProgressToken;
  /**
   * The MCP Protocol Version being used for this request.
   */
  "io.modelcontextprotocol/protocolVersion": string;
  /**
   * Identifies the client software.
   */
  "io.modelcontextprotocol/clientInfo": Implementation;
  /**
   * Capabilities of the client for this specific request.
   */
  "io.modelcontextprotocol/clientCapabilities": ClientCapabilities;
  /**
   * The desired log level for this request.
   */
  "io.modelcontextprotocol/logLevel"?: LoggingLevel;
}
```

Field semantics:

- `"io.modelcontextprotocol/protocolVersion"` : `string` — the MCP Protocol Version. **Required.** See the Protocol Version section above for negotiation details.
- `"io.modelcontextprotocol/clientInfo"` : `Implementation` — identifies the client software. **Required.** The `Implementation` schema requires `name` and `version`; other fields are optional.
- `"io.modelcontextprotocol/clientCapabilities"` : `ClientCapabilities` — the client's capabilities for this request. **Required.**
- `"io.modelcontextprotocol/logLevel"` : `LoggingLevel` — the desired log level for this request. **Optional.** If absent, the server **MUST NOT** send any log notifications for this request. The client opts in to log messages by explicitly setting a level. Replaces the `logging/setLevel` RPC.

Roots are intentionally not included as a per-request `_meta` field. Servers that need the client's roots **MUST** request them via the MRTR `ListRootsRequest` mechanism (see SEP-2322), which avoids putting potentially large root lists on every request and follows the "pay as you go" principle.

A request missing any required field is malformed; the server **MUST** reject it with `INVALID_PARAMS` (and `400 Bad Request` for HTTP).

Response Streaming

These declared capabilities govern what the server may include in the response stream. SEP-2322 (MRTR) defines how server-to-client interactions are embedded inline within responses via `IncompleteResult`; this SEP specifies that those interactions are governed by the per-request `clientCapabilities` declared in `RequestMetaObject`.

For HTTP, any request's response **MAY** be delivered as an SSE stream (`Content-Type: text/event-stream`) instead of a single JSON object. Only notifications (e.g., `notifications/progress` , `notifications/message`) flow as independent messages on this stream, followed by the final result. Server-to-client interactions (sampling, elicitation, listRoots) are **not** sent as independent requests — they are embedded as input requests inside an `IncompleteResult` returned from specific request paths (e.g., `CallTool` , `GetPrompt` , `ListResources`). The client satisfies the input requests and retries the original request.

Request Cancellation

How a client cancels an in-flight request depends on the transport:

- **HTTP.** Closing the SSE response stream **MUST** be treated by the server as cancellation of that request. Because each request has its own response stream, the transport-level disconnect is unambiguous.
- **STDIO.** The client **MUST** send a `notifications/cancelled` notification referencing the request ID. STDIO has a single shared channel, so there is no per-request stream to close.

Servers **SHOULD** stop work on a cancelled request as soon as practical and **MUST NOT** send any further messages for it.

Resumable Streams Are Removed

Because connection drops now implicitly cancel a request, resumable SSE streams (via `Last-Event-ID` reconnection) are removed. They contradict the stateless-by-default paradigm: resuming would require the server to retain per-request state across connection failures.

Workloads that need durability or resumability **MUST** use the tasks primitive instead, which provides explicit mechanisms for fetching results after a connection drop.

Missing Required Capabilities

A server **MUST NOT** rely on capabilities the client has not declared. If processing a request requires a capability the client did not declare in its `clientCapabilities`, the server **MUST** return a JSON-RPC error specifying the missing capabilities. For HTTP, the response status code MUST be `400 Bad Request`.

```
export const MISSING_REQUIRED_CLIENT_CAPABILITY = -32021;

export interface MissingRequiredClientCapabilityError extends Omit<
  JSONRPCErrorResponse,
  "error"
> {
  error: Error & {
    code: typeof MISSING_REQUIRED_CLIENT_CAPABILITY;
    data: {
      /**
       * The capabilities the server requires from the client
       * to process this request.
       */
      requiredCapabilities: ClientCapabilities;
    };
  };
}
```

`subscriptions/listen` RPC

This SEP introduces a new `subscriptions/listen` RPC that replaces the previous HTTP GET endpoint and ensures consistent behavior between HTTP and STDIO. A client uses it to open a long-lived channel for receiving notifications outside the context of a specific request.

The HTTP GET endpoint used by Streamable HTTP for server-to-client messages is **removed** in this version of the protocol. All communication uses POST.

Per SEP-2260, only notifications (not requests) flow on this channel; server-initiated requests use MRTR (see Response Streaming above) and are scoped to a specific client request.

Request Schema

```
export interface SubscriptionsListenRequest extends Request {
  method: "subscriptions/listen";
  params: {
    _meta: {
      "io.modelcontextprotocol/protocolVersion": string;
      "io.modelcontextprotocol/clientInfo": Implementation;
      "io.modelcontextprotocol/clientCapabilities": ClientCapabilities;
      // ... other meta fields
    };

    /**
     * The notifications the client wants to receive on this stream.
     * Each notification type is opt-in; the server **MUST NOT** send
     * notification types the client has not explicitly requested here.
     */
    notifications: {
      /**
       * If true, receive notifications/tools/list_changed.
       */
      toolsListChanged?: boolean;

      /**
       * If true, receive notifications/prompts/list_changed.
       */
      promptsListChanged?: boolean;

      /**
       * If true, receive notifications/resources/list_changed.
       */
      resourcesListChanged?: boolean;

      /**
       * Subscribe to notifications/resources/updated for specific
       * resource URIs. Replaces the resources/subscribe RPC.
       */
      resourceSubscriptions?: string[];
    };
  };
}
```

The `notifications` field is **required** and the client **MUST** explicitly opt in to each notification type it wants to receive. If a field within `notifications` is omitted (or set to `false`), the server **MUST NOT** send notifications of that type.

Acknowledgment Notification

The server sends this notification first to acknowledge that the subscription has been established. The subscription is long-lived and has no natural "completion result"; it ends when:

- the client explicitly cancels it (closing the SSE stream on HTTP, or sending `notifications/cancelled` on STDIO);
- the underlying connection is closed (HTTP timeout, TCP disconnect, STDIO process exit); or
- the server tears it down (e.g., shutdown), in which case it **MUST** close the SSE stream (HTTP) or send `notifications/cancelled` referencing the subscription's request ID (STDIO).

```
export interface SubscriptionsAcknowledgedNotification extends Notification {
  method: "notifications/subscriptions/acknowledged";
  params: {
    /**
     * The notification subscriptions the server has agreed to honor.
     * Only includes notification types the server actually supports.
     * If the client requested an unsupported notification type
     * (e.g., promptsListChanged when the server has no prompts),
     * it is omitted from this set.
     */
    notifications: {
      toolsListChanged?: boolean;
      promptsListChanged?: boolean;
      resourcesListChanged?: boolean;
      resourceSubscriptions?: string[];
    };
  };
}
```

Multiple Concurrent Subscriptions

A client **MAY** have multiple active subscriptions concurrently (e.g., one listening for tools-list changes, another for resource updates). Each subscription is identified by the JSON-RPC request ID of its `SubscriptionsListenRequest`.

To allow STDIO clients to demultiplex notifications belonging to different subscriptions on the single shared channel, every notification delivered as part of an active subscription **MUST** include the subscription's request ID in `_meta`:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tools/list_changed",
  "params": {
    "_meta": {
      "io.modelcontextprotocol/subscriptionId": "<original listen request id>"
    }
  }
}
```

This same correlation pattern applies to other server-to-client notifications that need to be associated with a specific request, such as `notifications/progress` (which uses the originating request's ID).

Stopping a Subscription

- **HTTP.** Closing the SSE response stream stops the subscription.

- **STDIO.** The client sends `notifications/cancelled` referencing the listen request's ID. The server **MUST** stop sending notifications for that subscription.

Transport Behavior

HTTP. The client sends `SubscriptionsListenRequest` via `POST`. The server's response is an open SSE stream (`Content-Type: text/event-stream`), and the first JSON-RPC message on this stream **MUST** be a `SubscriptionsAcknowledgedNotification`.

STDIO. The client sends `SubscriptionsListenRequest` at any time. The server **MUST** acknowledge it by sending a `SubscriptionsAcknowledgedNotification`. Subsequent notifications flow on the bidirectional STDIO channel, each tagged with the subscription's request ID as described above. If the connection is terminated (e.g., the server crashes and restarts), the client **MUST** re-send `SubscriptionsListenRequest` to re-establish its subscriptions.

Deprecated and Removed RPCs

To simplify the protocol and align with the move to per-request capabilities, the following RPC methods and notifications are removed:

- `initialize` / `notifications/initialized`: The initialization handshake is removed. Version negotiation is handled per-request via `MCP-Protocol-Version` headers and `_meta` fields. Capability discovery is handled by `server/discover`.
- `logging/setLevel`: Removed. The log level is now specified per-request via the `'io.modelcontextprotocol/logLevel'` `_meta` field. There is no replacement RPC.
- `roots/list`: Removed as a top-level server-to-client RPC. Servers that need the client's roots **MUST** request them via the MRTR `ListRootsRequest` mechanism (see SEP-2322).
- `notifications/roots/list_changed`: Removed. Roots are fetched on demand via MRTR, so there is no need for a change notification.
- `resources/subscribe` / `resources/unsubscribe`: These methods are removed. Resource subscriptions are inherently stateful — the server must remember which resources each client has subscribed to. Instead, clients declare the resources they want updates for in the `notifications` param of the `subscriptions/listen` request. The server sends `notifications/resources/updated` on the listen stream for matching resources.
- `ping`: Removed in **both directions**. Server-to-client ping is removed because servers can no longer independently send requests. Client-to-server ping is also removed because any normal RPC call already proves server liveness, and transport-layer mechanisms (HTTP keep-alives, SSE comments, STDIO process status) handle connection-health checks more appropriately.

Rationale

Stateless-First by Default

The primary design decision of this SEP is to remove the mandatory initialization handshake, making stateless interaction the default model for the protocol. This choice is rooted in the "pay as you go" principle and the desire to align MCP with modern, cloud-native architecture. By making the simplest interaction model the default, we lower the barrier to entry and reduce implementation complexity for the most common use cases. This immediately enables straightforward horizontal scaling and improves resilience, as any request can be handled by any server instance.

Alternative Considered: Optional Handshake

An alternative we considered was to keep the existing stateful handshake but make it optional. In this model, a client could choose to either perform the handshake to establish a persistent session or skip it and send self-contained requests.

Why it was rejected:

Supporting two parallel interaction models would have dramatically increased the complexity of the protocol and every implementation. Servers and clients would need to build, test, and maintain two separate logic paths, leading to a larger surface area for bugs. It also violates the design principle of having one clear, obvious way to perform a core function. By making a clean break, we ensure the entire ecosystem can move forward and benefit from a simpler, more scalable, and more robust foundation.

Explicit Session Management

This proposal originally included dedicated `sessions/create` and `sessions/delete` RPCs to manage the lifecycle of a logical session.

Session management is now addressed separately by SEP-2567, which proposes removing sessions entirely and replacing them with explicit state handles. This aligns with the sessions-vs-sessionless decision made by the Core Maintainers.

Separation of Concerns

A core principle of this proposal is the "unbundling" of the monolithic initialization handshake into a suite of discrete, single-purpose RPCs. The original handshake mixed the concerns of protocol negotiation and capability discovery into a single, complex interaction. The new design explicitly separates these:

- **Discovery**: Handled exclusively by `server/discover`.
- **Capabilities**: Handled on a per-request basis via the `_meta` field or the `subscriptions/listen` RPC.

The rationale for this is to create a more modular, flexible, and understandable protocol. Each component now has a single, well-defined responsibility. This allows clients to use only the parts of the protocol they need, adhering to our "pay as you go" principle.

Alternative Considered: A Monolithic Handshake

We could have kept a single, monolithic handshake RPC and simply added more parameters and complex logic to it to support the stateless-first model.

Why it was rejected:

A single, do-it-all RPC is difficult to implement, test, and evolve. It forces all clients, even the simplest ones, to be aware of the protocol's most complex features. By separating these concerns, we've made the protocol easier to learn and implement correctly, while also making it more flexible and extensible for the future.

Backward Compatibility

While this proposal attempts to preserve existing functionality and use-cases, this proposal introduces a **fundamental, backward-incompatible change**. Thus, it will require a new version of the protocol.

Supporting Multiple Versions

While this SEP removes the `initialize` handshake, a server that wishes to support both old and new clients **MAY** do so. Such a server can continue to implement the old `initialize` RPC to handle legacy clients, while also exposing the new stateless RPCs (`server/discover`, etc.) for updated clients.

Both servers and clients should be able to handle changes in the versions appropriately. Two example scenarios are outlined below, where vPrev indicates the version prior to the SEP, and vAfter indicates a version after it.

Client (supporting vPrev) → Server (vPrev, vPost)

1. Client sends initialization
2. Server supports vPrev, so initialization is returned per spec
3. Client and server communicate per `vPrev`.

Client (supporting vPrev, vPost) → Server (vPrev)

For HTTP, the client may attempt any vPost request (e.g., `tools/list` with the MCP Protocol Version header). The server returns `400 Bad Request` (or `Unsupported protocol version`); the client falls back to vPrev (and performs initialization) for future requests.

For STDIO, the client cannot rely on a per-request error to detect the server's version. A client that supports both a vPost (which does not require initialization) **and** a legacy version that does require `initialize` **SHOULD** probe with `server/discover` first to determine which to use:

1. Client sends `server/discover` with the MCP Protocol Version `_meta` field set to its preferred vPost.
2. If the server supports vPost (or any vPost-style version the client also supports), the client uses the discovered version for subsequent requests.
3. If the server returns `Unsupported protocol version` or `Method not found`, the client falls back to its supported legacy version and performs the `initialize` handshake.

A client that supports only vPost-style versions has no need to probe — it simply uses its preferred version and handles `Unsupported protocol version` errors normally.

Security Implications

Without a session handshake, every request must be independently authenticated and authorized. Implementations **MUST** ensure that authentication is not bypassed by the removal of the initialization phase.

Beyond per-request authentication, this proposal does not introduce additional security concerns.

Reference Implementation

// TODO

FAQ

What is protocol level statelessness?

Wikipedia defines a stateless protocol as:

> A stateless protocol is a communication protocol in which the receiver must not retain session state from previous requests. The sender transfers relevant session state to the receiver in such a way that every request can be understood in isolation, that is without reference to session state from previous requests retained by the receiver.

This does NOT mean that you can't build stateful applications on top of a stateless protocol. HTTP is an example of a stateless protocol, which most of the web is built on today. However it does mean that the state cannot exist *in the protocol itself*, and should instead specify the state in the request (or failing that, a reference to the state for the server or client to track).

Does this make MCP a fully stateless protocol?

Not entirely (hence 'by default'). Depending on your interpretation of "requests", the SSE streams mentioned (both client-initiated and server-initiated) tend to have multiple requests within a context of a stream. However, these streams are constrained to a single HTTP request and optional to use, meaning that the complexity is both constrained and optional to use when the situation requires it.

Why is it important for STDIO to be stateless as well?

The transport MCP is using should be an implementation detail only. If one version of a protocol supports functionality that doesn't cleanly map over to another version of the protocol, they are really two different protocols.

This makes it easy for developers to switch their services from one transport to another without needing to make significant changes to the behavior of their applications, and easier to proxy between different transports correctly. Otherwise, there will continue to be feature gaps and division between these different implementations, leading to both confusion and incompatibility.

How does `server/discover` relate to the MCP Server Card?

The `server/discover` RPC overlaps with the MCP Server Card proposal, which defines a `.well-known/mcp.json` document for HTTP-based discovery. Both mechanisms are intentionally retained: the Server Card is well-suited to HTTP (no auth required, cacheable, indexable) while `server/discover` provides a unified RPC interface that works consistently across HTTP and STDIO transports. The two should be aligned on content where applicable.

Open Questions

What belongs in `_meta` vs. as a top-level protocol field?

This SEP places several previously-handshake-negotiated values (`protocolVersion` , `clientInfo` , `roots` , `logLevel` , `clientCapabilities`) into per-request `_meta` fields under the `io.modelcontextprotocol/` namespace. This follows the spec's allowance for "purpose-specific metadata" reserved by definitions in the schema.

However, this risks overloading `_meta` over time — at what point do we add top-level fields again? One possible distinction: required protocol-level fields (e.g., `protocolVersion`) might better live as top-level fields, while optional or extension-provided values stay in `_meta` . This question deserves broader discussion before this SEP is finalized.

Should `clientInfo` be part of `ClientCapabilities`?

Currently, `clientInfo` (`Implementation` type) and `clientCapabilities` (`ClientCapabilities` type) are separate fields. In a per-request model, having a single field for all client metadata would reduce overhead. However, `clientInfo` serves a different purpose (identity/UI) than capabilities (feature negotiation). Should `clientInfo` be folded into `ClientCapabilities`, remain a separate per-request `_meta` field, or be handled through a different mechanism entirely (e.g., only sent via `subscriptions/listen`)?

Changes since SEP became Final

This SEP is preserved as a historical record of what was accepted. The list below tracks changes made to the specification after this SEP reached Final status. Refer to the current specification for the authoritative, up-to-date requirements.

- **Client identity became optional request metadata.** #3002 made `io.modelcontextprotocol/clientInfo` optional. Clients **SHOULD** include it on every request unless specifically configured not to do so.
- **Server identity moved to optional result metadata.** #3002 introduced `io.modelcontextprotocol/serverInfo` in result `_meta` and removed the top-level `DiscoverResult.serverInfo` field to avoid duplicate representations. Servers **SHOULD** include this metadata on every result unless specifically configured not to do so.
- **Subscriptions gained a graceful completion result.** #2953 defined a `subscriptions/listen` result for server-initiated graceful closure, replacing the SEP's statement that a subscription has no natural completion result. Servers **SHOULD** send this result before closing the stream.

SEP-2577 Deprecate Roots, Sampling, and Logging

Final · Standards Track · Created 2026-04-14

- **Status**: Final
- **Type**: Standards Track
- **Created**: 2026-04-14
- **Author(s)**: Kurtis Van Gent (@kurtisvg)
- **Sponsor**: @kurtisvg
- **PR**: #2577

> **Note**: This SEP is predicated on a hypothetical SEP where MCP considers a specification version supported for one year past its original release date. The deprecation timeline described here assumes that policy is in place.

Abstract

This SEP deprecates the following core protocol features:

- **Roots** (`roots/list`, `notifications/roots/list_changed`)
- **Sampling** (`sampling/createMessage`, `ClientCapabilities.tasks.requests.sampling`)
- **Logging** (`logging/setLevel`, `notifications/message`)

These features are deprecated starting in the specification version that includes this SEP (expected June 2026). They will continue to be fully functional in all specification versions released within one year of that version's release.

Each of those subsequent versions will in turn support the features for one year after its own release, assuming the one-year-per-version support policy proposed in a separate SEP. This provides implementations with an extended migration window before the features are fully removed.

During the deprecation period, wire-level behavior is unchanged. No types are removed, no capability negotiation changes, and no existing implementations break. The deprecation serves as a signal to the ecosystem to stop building on these features and to plan for their eventual removal.

Motivation

The MCP specification aims to remain minimal and focused. Features that see low adoption, overlap with existing alternatives, or impose disproportionate implementation burden relative to their value are candidates for removal. Keeping such features in the core specification increases the burden for every client and server, slows protocol evolution, and makes the specification harder to learn. The following three features meet these criteria.

Deprecating these features was proposed during a recent core contributor meeting. This SEP formalizes that proposal with a concrete implementation plan. See discussion #2536.

Roots

Roots provides "informational guidance" about which directories or files a server should operate on. In practice:

- **Low adoption**: Few clients implement roots support, and few servers rely on it. The feature support matrix shows limited client coverage.
- **Vague semantics**: The specification describes roots as informational — servers are not required to respect them, which reduces their utility.
- **Overlapping alternatives**: Working directory context can be provided through tool parameters, resource URIs, server configuration, or environment variables — all of which are more explicit.

Sampling

Sampling allows servers to request LLM completions from the client. While conceptually powerful, it has struggled with adoption:

- **Complex to implement**: Correct sampling implementation requires human-in-the-loop approval, model selection logic, security considerations, and (since SEP-1577) tool loop support. This complexity has contributed to low client adoption.
- **Low adoption**: The feature support matrix shows that few clients support sampling, despite the feature being available since the November 2024 specification.
- **Direct alternatives**: Servers that need LLM capabilities can integrate directly with LLM provider APIs, giving them full control over model selection, parameters, and streaming.

Logging

Logging allows servers to send structured log messages to clients via the protocol:

- **Overlapping infrastructure**: Standard logging mechanisms (stderr for stdio transports, OpenTelemetry for structured observability) are mature, widely adopted, and better suited to logging than an application-protocol channel.
- **Low value relative to complexity**: Adding log message types, severity levels, and the `logging/setLevel` request to the core specification increases the implementation surface for all clients and servers.

Specification

Overview of changes

1. Mark deprecated features with `@deprecated` annotations in the schema
2. Add deprecation notices to feature documentation pages
3. No wire-level protocol changes during the deprecation period

Schema changes

Add `@deprecated` JSDoc annotations to the following items in `schema/draft/schema.ts`. No types, interfaces, or union members are removed.

Deprecated capabilities

Capability	Location
`ClientCapabilities.roots`	Client capability for listing roots
`ClientCapabilities.sampling`	Client capability for LLM sampling
`ClientCapabilities.tasks.requests.sampling`	Task-augmented sampling sub-capability
`ServerCapabilities.logging`	Server capability for log messages

Deprecated types — Roots

Type	Description
`Root`	Represents a root directory or file
`ListRootsRequest`	Server-to-client request for `roots/list`
`ListRootsResult`	Result containing roots array
`ListRootsResultResponse`	JSON-RPC response wrapper
`RootsListChangedNotification`	Client notification when roots change

Deprecated types — Sampling

Type	Description
`CreateMessageRequestParams`	Parameters for `sampling/createMessage`
`CreateMessageRequest`	Server-to-client request for sampling
`CreateMessageResult`	Result from a sampling request
`CreateMessageResultResponse`	JSON-RPC response wrapper
`SamplingMessage`	A message in a sampling conversation
`SamplingMessageContentBlock`	Content block union for sampling messages
`ToolChoice`	Controls model tool selection during sampling
`ToolUseContent`	Tool use content block in sampling messages
`ToolResultContent`	Tool result content block in sampling messages
`ModelPreferences`	Server preferences for model selection
`ModelHint`	Hints for model selection

Deprecated types — Logging

Type	Description
`LoggingLevel`	Syslog severity level enum
`SetLevelRequestParams`	Parameters for `logging/setLevel`
`SetLevelRequest`	Client-to-server request to set level
`SetLevelResultResponse`	JSON-RPC response wrapper
`LoggingMessageNotificationParams`	Parameters for log message notification
`LoggingMessageNotification`	Server-to-client log message

Annotation format

Each deprecated item SHOULD receive a JSDoc `@deprecated` tag with a brief explanation:

```
/**
 * Present if the client supports listing roots.
 *
 * @deprecated Deprecated as of this specification version. Will be included
 * in all versions released within one year, then may be removed.
 */
roots?: {
  listChanged?: boolean;
};
```

Union types

The following union types reference deprecated types but MUST NOT be modified during the deprecation period. They will be updated when the deprecated types are removed:

- `ClientNotification` (includes `RootsListChangedNotification`)
- `ClientResult` (includes `CreateMessageResult`, `ListRootsResult`)
- `ServerRequest` (includes `CreateMessageRequest`, `ListRootsRequest`)
- `ServerNotification` (includes `LoggingMessageNotification`)

Documentation changes

Add a deprecation warning block at the top of each feature's documentation page, after the title:

`docs/specification/draft/client/roots.mdx`:

```
<Warning>
**Deprecated**: The Roots feature is deprecated as of this specification
version. It will remain fully functional in all specification versions released
within one year of the <YYYY-MM-DD> release. Each of those versions will
continue to support it for one year after its own release.
</Warning>
```

`docs/specification/draft/client/sampling.mdx`:

```
<Warning>
**Deprecated**: The Sampling feature is deprecated as of this specification
version. It will remain fully functional in all specification versions released
within one year of the <YYYY-MM-DD> release. Each of those versions will
continue to support it for one year after its own release.
</Warning>
```

`docs/specification/draft/server/utilities/logging.mdx`:

```
<Warning>
**Deprecated**: The Logging feature is deprecated as of this specification
version. It will remain fully functional in all specification versions released
within one year of the <YYYY-MM-DD> release. Each of those versions will
continue to support it for one year after its own release.
</Warning>
```

Capability negotiation

During the deprecation period, capability negotiation is **unchanged**:

- Clients and servers that support deprecated features SHOULD continue to declare the corresponding capabilities.
- Implementations that encounter deprecated capabilities MUST still handle them correctly.
- Implementations SHOULD emit a warning (e.g., in logs or developer tooling) when deprecated capabilities are negotiated.
- New implementations SHOULD NOT add support for deprecated features unless needed for backward compatibility with existing counterparts.

Timeline

- **Deprecated**: In the next specification release (currently planned for June 2026).
- **Included in subsequent releases**: All specification versions released within one year of this version's release MUST continue to include these features as deprecated.
- **Per-version support**: Each version that includes these features will support them for one year after that version's release, per the one-year-per-version support policy proposed in a separate SEP.
- **Removal**: Specification versions released more than one year after this version's release MAY remove these features entirely.

Rationale

Why deprecate rather than move to extensions?

These features are already implemented in many clients and servers. The extensions mechanism (SEP-2133) specifies that unless an extension is provided, implementations must behave as if the extension is not present. Retrofitting this logic into existing SDKs — especially across multiple protocol versions — would be complex and error-prone. Deprecation followed by removal is less disruptive: implementations can continue using the features as-is during the transition period, then simply stop when the features are removed.

Why deprecate rather than remove immediately?

While adoption of these features is low, they are still in use. Removing them immediately would cause unnecessary churn and disruption for users, client and server owners, and SDK builders. A deprecation window minimizes this impact by giving the ecosystem time to migrate at its own pace.

Why these three features specifically?

These were identified during a core contributor meeting as the features with the weakest adoption-to-complexity ratio. Each has viable alternatives outside the protocol, and none are critical to the core resource/tool/prompt interaction model that defines MCP. See discussion #2536.

Backward Compatibility

During the deprecation period, there are **no backward compatibility issues**. All deprecated features continue to work identically. No wire-level changes are introduced.

After removal (in specification versions released more than one year after this version):

- Implementations negotiating an older protocol version that includes these features will still have access to them through that version's schema.
- Implementations negotiating a version that has removed these features will no longer have access to them.

Security Implications

Deprecating these features has a **net positive** effect on security:

- **Sampling** is the most security-sensitive of the three. It allows servers to request LLM completions through the client, which creates attack surface for prompt injection and data exfiltration. Removing it reduces this risk.
- **Roots** exposes information about the client's filesystem to servers. Removing it reduces the risk of servers using root information to attempt directory traversal or access files outside intended boundaries.
- **Logging** has minimal security implications, but removing it simplifies the protocol surface area.

No new security concerns are introduced by deprecation.

Reference Implementation

No reference implementation is required. This SEP only marks existing functionality as deprecated — no new protocol behavior is introduced.

SEP-2596 Specification Feature Lifecycle and Deprecation Policy

Final · Process · Created 2026-04-17

- **Status**: Final
- **Type**: Process
- **Created**: 2026-04-17
- **Author(s)**: Den Delimarsky (@localden)
- **Sponsor**: @localden
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2596

Abstract

This SEP defines a lifecycle for individual features within the Model Context Protocol specification, separate from the revision lifecycle of the specification document itself. It introduces three feature states (Active, Deprecated, Removed), the criteria and procedure for moving between them, a minimum window between deprecation and removal, and the documentation required at each transition. The goal is a predictable timeline that SDK authors and implementers can plan migrations against when protocol surface area is retired.

Motivation

The specification has already retired or signaled retirement of several features, but each case has been handled ad hoc:

- The HTTP+SSE transport is described as "deprecated" in the Streamable HTTP backwards-compatibility guidance, with no stated removal date.
- The `includeContext` values `"thisServer"` and `"allServers"` are labeled "soft-deprecated" in `sampling/createMessage` and in `schema.ts`, with the note that they "may be removed in future spec releases."
- JSON-RPC batching was added in revision `2025-03-26` and removed in `2025-06-18`, a single release later, with no deprecation period.
- Open proposals such as consolidating `Resource` and `ResourceTemplate` (#1540) and deprecating roots, sampling, and logging (SEP-2577) would each retire existing surface area but have no process to follow.

This inconsistency has costs. Implementers cannot tell whether "deprecated" and "soft-deprecated" mean different things, or how long either state lasts before removal. Community questions such as discussion #2177 (asking when the SSE transport will actually be removed) have no policy to point to. At the NYC maintainer meeting, large implementers described indefinite support for past protocol versions as "corrosive tech debt." The Stability over velocity design principle observes that "removing from [the spec] is nearly impossible" but offers no path for the cases where removal is warranted.

The Core Maintainers agreed at the April 1, 2026 meeting that MCP needs "a formal versioning status and a defined deprecation cycle" with "direction agreed, mechanics TBD." This SEP proposes those mechanics.

Specification

Scope

This policy governs **features** of the MCP core specification: protocol messages, capabilities, transports, schema types, and normative behavioral requirements. It does not govern the independent lifecycle of SDK-specific APIs, registry policies, or the revision lifecycle of the specification document itself (Draft, Current, Final), which is defined in the versioning guide.

Note that "Final" is used in two senses in this document: a specification *revision* is Final when superseded by a later one (per the versioning guide), and a *SEP* reaches Final when its status advances per the SEP guidelines. Context disambiguates; where it does not, this document writes "the SEP reaches Final" or "Final revision" explicitly.

Feature states

A specification feature is in exactly one of three states:

State	Meaning	Implementer expectation
Active	The feature is part of the Current specification revision with no planned removal.	Implement per the feature's normative requirements.
Deprecated	The feature remains in the specification but is scheduled for removal. A migration path is documented (see below).	New implementations SHOULD NOT adopt the feature. Existing implementations SHOULD migrate before the earliest removal date.
Removed	The feature has been deleted from `draft` and will be absent from the next Current revision. It remains documented in the Final revision it last appeared in.	Implementations targeting that next Current revision MUST NOT depend on the feature.

The term "soft-deprecated" is retired. Existing uses in the specification are reclassified as Deprecated under this policy (see Transition).

Removal from the specification does not oblige an SDK to drop the feature from releases that continue to support an earlier revision in which it was Active or Deprecated; that timeline is governed by the SDK's own revision-support policy (see Open Questions).

A Deprecated feature MAY be restored to Active by a SEP that supersedes the deprecation SEP and documents the changed circumstances. Restoration follows the same approval path as deprecation. If the feature is later deprecated again, the minimum deprecation window in Deprecating a feature is measured afresh from the revision in which the new deprecation takes effect.

Deprecating a feature

A feature MAY be proposed for deprecation when at least one of the following holds:

- It has been superseded by another feature that covers the same use cases.
- It presents a security, privacy, or interoperability risk that cannot be mitigated in place.

- Ecosystem telemetry or SDK maintainer consensus indicates negligible adoption relative to its maintenance cost.

Deprecation is a specification change and therefore requires a SEP per the SEP guidelines. The deprecation SEP MUST:

1. Identify the feature by name and link to its definition in `schema.ts` (where applicable) and the specification prose.
2. State the rationale against the criteria above.
3. Document the migration path, or state explicitly that none is required. If the migration path names a replacement feature, that feature MUST be Active in the revision in which the deprecation takes effect; the replacement and the deprecation MAY land in the same revision. A feature is not deprecated under this policy while its documented replacement is still only in `draft`.
4. Specify the **minimum deprecation window**: the number of months, at least twelve, that the feature MUST remain Deprecated before it is eligible for removal. The window is measured from the release of the specification revision in which the feature is first marked Deprecated, not from the date the SEP reaches Final. The feature becomes eligible for removal in the first specification revision released as Current on or after the window elapses; that point is the feature's **earliest removal**.

When the deprecation SEP reaches Final the deprecation is scheduled: the following changes land in the draft specification (`schema/draft/` and `docs/specification/draft/`). The feature becomes Deprecated when the revision carrying these changes is released as Current under the versioning guide, and the minimum deprecation window is counted from that release. Anchoring the clock to the revision release means every feature deprecated in the same revision shares one earliest removal rather than each carrying a date derived from when its own SEP happened to land.

- The feature's entry in `schema.ts` gains a `@deprecated` JSDoc tag referencing the deprecation SEP and the revision in which the deprecation takes effect.
- The specification prose for the feature gains a deprecation notice with the same information.
- The `changelog.mdx` for that revision gains an entry under a "Deprecated" heading. This SEP introduces "Deprecated" and "Removed" as standing changelog headings alongside the existing Major/Minor/Other groupings.
- The feature is added to the deprecated registry with its deprecation SEP, the revision in which it became Deprecated, its migration path, and its earliest removal.

The deprecated registry

`docs/specification/draft/deprecated.mdx` is a single page listing every feature currently in the Deprecated state. It is the canonical answer to "what is on its way out, and by when," so that an implementer does not have to reconstruct that picture from deprecation entries spread across revision changelogs. Each row records the feature, its deprecation SEP, the revision in which it became Deprecated, the documented migration path, and its earliest removal. A deprecation adds a row; a removal moves the row to a Removed section of the same page with a link to the changelog entry, so the page also serves as the historical record. The registry carries no normative force of its own; it is a derived view kept consistent with the per-feature notices and changelog entries, which are the normative records.

Tier 1 SDK obligations

A feature lifecycle is only as effective as the implementations that surface it to consumers. The specification artifacts above record that a feature is Deprecated; Tier 1 SDKs (per SEP-1730) deliver that record to the implementers who would otherwise discover the removal by breakage. Once the revision in which a feature becomes Deprecated is released as Current, Tier 1 SDKs:

- MUST mark the corresponding API surface deprecated using the language's native mechanism (for example `@Deprecated` in Java, `[Obsolete]` in .NET, `@deprecated` JSDoc in TypeScript, the `Deprecated:` doc convention in Go) in their next release, referencing the deprecation SEP and the earliest removal date

where the mechanism permits. The marker applies to the SDK API surface and is not conditioned on the specification revision a consumer targets; surfacing it to consumers still on an earlier revision is intentional forward signal.
- SHOULD emit a runtime warning when a deprecated feature is exercised, using the language's idiomatic mechanism (for example Python's `DeprecationWarning`, Node.js's `process.emitWarning`, or a configurable logger). A runtime warning reaches developers who never read API documentation and is an observable signal a conformance test can assert against.

These obligations are conformance criteria for Tier 1 status. A Tier 1 SDK that persistently fails to surface a Deprecated feature is subject to the Tier Relegation Process in SEP-1730.

Removing a feature

1. Once a feature is set for removal, the removal is executed at the discretion of the Core Maintainers after the minimum deprecation window has elapsed, during release preparation, under the governance decision process. Removal does not require its own SEP. Before removing a feature the Core Maintainers MUST confirm that the migration target named in the deprecation SEP, if any, is still Active.
2. A SEP is required for any other change to a deprecation or removal, for example extending or shortening the timeline (Expedited removal) or restoring the feature to Active (Feature states).

Note that features may remain Deprecated, without removal, for much longer than the minimum deprecation window.

SDKs implement deprecation as part of the SDK Tiering System (see Tier 1 SDK obligations); removal imposes no additional requirements on SDK maintainers.

When a removal decision is taken, the feature is deleted from `schema/draft/schema.ts` (where present) and the draft specification prose; `changelog.mdx` for that revision gains an entry under the "Removed" heading that links to the deprecation SEP and the last Final revision in which the feature was present; and the feature's registry row moves to the Removed section with a link to that changelog entry.

Expedited removal

The twelve-month floor MAY be shortened when the feature presents an active security risk, meaning a vulnerability with a published security advisory or documented in-the-wild exploitation for which no in-place mitigation exists. Shortening the window requires Core Maintainer approval under the governance decision process, recorded in the deprecation SEP or, where the risk surfaces after that SEP is already Final, in a short expedited-removal SEP that references it. The shortened window MUST still provide at least ninety days between the feature becoming Deprecated and its earliest removal.

Roles

Action	Who
Propose deprecation, extension, or restoration	Any contributor, per the SEP process
Sponsor	A Maintainer or Core Maintainer, per the SEP process
Approve a deprecation SEP	Core Maintainers, per the governance decision process
Decide a removal during release preparation	Core Maintainers, per the governance decision process
Approve an extension or restoration SEP	Core Maintainers, per the governance decision process
Approve expedited removal	Core Maintainers, per the governance decision process

As with all Core Maintainer decisions, Lead Maintainers retain veto authority over each of the approvals above, per the governance roles definition.

Transition

Two features were already described as deprecated in the specification before this policy existed (see Motivation). When this SEP reaches Final they are classified as Deprecated and seeded into the registry; the deprecation-SEP requirements in Deprecating a feature are not applied retroactively. The deprecation decision in each case predates this policy; this section records it under the new vocabulary so the terms "deprecated" and "soft-deprecated" carry a single defined meaning going forward.

Both features were publicly deprecated well over twelve months before this SEP, so the minimum deprecation window has in practice already been served; re-anchoring their clock to a future revision release would restart a window the ecosystem has already had. Each is therefore given a three-month grace period from the date this SEP reaches Final before it is eligible for removal, matching the floor the Expedited removal clause sets for the shortest permissible window. Removal still follows Removing a feature: a Core Maintainer decision at release preparation, not an automatic event when the grace period ends.

Feature	Migration target	Earliest removal
HTTP+SSE transport	Streamable HTTP	Three months after this SEP is Final
`includeContext: "thisServer"` / `"allServers"`	Omit the field or use `"none"`	Follows Sampling (SEP-2577)

`includeContext` is a parameter of `sampling/createMessage`. SEP-2577 deprecates the Sampling feature as a whole; the two affected `includeContext` values follow that feature's deprecation schedule rather than carrying an independent removal clock, and are removed no later than Sampling itself.

This grandfathering applies only to features the specification already described as deprecated on the date this SEP reaches Final. Every subsequent deprecation follows Deprecating a feature in full, and removal of the

grandfathered features follows Removing a feature without exception.

When this SEP reaches Final the following land in `draft/` directly, with no separate implementation gate: the versioning guide is updated to reference this policy; `deprecated.mdx` is created seeded with the two features above; the "Deprecated" heading is added to `changelog.mdx` with both entries; and each feature gains the `@deprecated` schema annotation and prose notice described in Deprecating a feature. For `includeContext` the annotation is on the property as a whole, since per-value `@deprecated` tags are not expressible on a string-literal union; the HTTP+SSE transport has no `schema.ts` types and is annotated in the transport prose only.

Rationale

Why a separate state model from specification revisions?

The versioning guide already defines Draft, Current, and Final for specification *revisions*. Those states describe the editorial maturity of a whole document and say nothing about whether a given message or field within a Current revision is on its way out. The Kubernetes deprecation policy, the Node.js deprecation cycle, and IETF practice such as RFC 8996 (which deprecates TLS 1.0 and 1.1 within the TLS protocol family) all maintain feature-level deprecation rules alongside their release versioning for this reason.

Why a SEP to deprecate but not to remove?

The deliberation that needs community review is the decision to retire a feature and the choice of migration path; that is what the deprecation SEP carries. Once it reaches Final the project has committed to removal and fixed the earliest date, so carrying out that decision on schedule adds no new judgment and a second SEP for it is process for its own sake. The deliberate maintainer decision still exists as the release-preparation removal decision and its confirmations in Removing a feature, mirroring the tier advancement procedure in SEP-1730 where advancement is a maintainer decision rather than a timer expiring. A SEP is reserved for the cases that do change the committed outcome: extending the window, restoring the feature, or shortening the floor for a security risk. This keeps the process consistent with the SEP guidelines treating a change to protocol surface area as SEP-worthy while not demanding a SEP to ratify a change already made.

Why twelve months?

The NYC maintainer meeting floated a "one year supported plus one year deprecation" model and recorded reluctance to commit to longer windows given how quickly the agentic space is moving. The same discussion flagged even that model as a possible burden on SDK maintainers; this SEP keeps the twelve-month floor because removal is permissive rather than automatic (Removing a feature), so a feature stays Deprecated as long as the ecosystem needs rather than the SDKs racing the calendar. Measuring the window from the revision release rather than from the SEP date keeps it observable: it is the same clock SDK authors and implementers already track for the revision itself. Because the deprecation only takes effect when its revision is released, a replacement introduced in that same revision is proven over the twelve-month window itself; a separate prior revision is not required for that purpose. The window spans at least two of the six-month release cycles discussed at the same meeting: one for SDK maintainers to ship migration support and one for downstream adoption. Core Maintainers may leave a feature Deprecated for longer; twelve months is the minimum.

Relationship to SEP-1400 (Semantic Versioning)

SEP-1400 proposes replacing date-based revision identifiers with semantic versioning. The two proposals address different questions: SEP-1400 is about how revisions are numbered, and this SEP is about how features within a revision are retired. This SEP measures the removal window from a revision *release* rather than from a revision *identifier*, so it does not depend on the identifier scheme; it applies unchanged whether revisions are dated or semantically versioned.

Consensus

Direction was agreed at the NYC maintainer meeting (March 31, 2026) and confirmed at the April 1, 2026 Core Maintainer meeting, which recorded "formal versioning status and SDK deprecation cycle (direction agreed, mechanics TBD)." Community demand is visible in discussion #2177 (asking when SSE removal will happen) and discussion #1980 (asking to sunset a backwards-compatibility requirement that has outlived its purpose).

Backward Compatibility

This SEP introduces a process and does not change protocol behavior. The Transition section assigns a Deprecated state and an earliest removal to two features that are already informally deprecated. Neither had a stated removal date, so making the timeline explicit (a three-month grace period for features the ecosystem has already had more than a year to migrate away from) does not shorten any commitment implementers were given.

Security Implications

None identified. This is a governance change with no new protocol surface, transport, authentication flow, or trust boundary. A defined deprecation path has an indirect security benefit: it gives the project a predictable mechanism for retiring features that are later found to be unsafe, which is what the Expedited removal clause is for.

Reference Implementation

This SEP defines a process and has no reference implementation. The specification edits that apply the policy to the two existing informal deprecations are described in Transition and land directly in `draft/` when this SEP reaches Final.

Open Questions

- **Specification revision support window.** The NYC meeting also discussed how long Tier 1 SDKs must support a given specification *revision* (as distinct from a feature within one). That policy belongs in an amendment to SEP-1730, but it determines whether the deprecation window in this policy is observable in practice. If a Tier 1 SDK supports only the latest revision, a consumer that updates the SDK between releases can move directly from one that predates the deprecation to one that postdates the removal, never seeing the Deprecated marker in Tier 1 SDK obligations. Requiring Tier 1 SDKs to support all revisions released as Current within a trailing window at least equal to the twelve-month deprecation floor closes that gap. The SEP-1730 amendment should be pursued alongside this SEP.
- **Telemetry source for the "negligible adoption" criterion.** The policy permits deprecation on adoption grounds, but the project has no shared telemetry today. Until one exists, this criterion relies on SDK maintainer attestation.
- **Feature maturity tiers.** This SEP applies a uniform twelve-month floor to every Active feature. The Kubernetes deprecation policy uses alpha/beta/GA tiers with shorter windows for less mature features, which would have allowed the JSON-RPC batching reversal cited in Motivation without a year-long deprecation. Whether MCP should adopt an Experimental tier with a shorter or zero window is left for a follow-up SEP.
- **Wire-level deprecation signal.** Tier 1 SDK obligations puts the deprecation warning into official SDKs; implementers that do not use one and do not read the changelog still receive no warning before removal. A wire-level signal (for example a `_meta` deprecation field on responses, comparable to the Kubernetes

`Warning` header) would close that gap but is a Standards Track change outside the scope of this Process SEP.

SEP-2640 Skills Extension

Final · Extensions Track · Created 2026-04-23

- **Status**: Final
- **Type**: Extensions Track
- **Created**: 2026-04-23
- **Author(s)**: Peter Alexander (@pja-ant), Ola Hungerford (@olaservo), Sambhav Kothari (@sambhav), Aditya Kumar (@aditya-scio), on behalf of the Skills Over MCP Working Group
- **Sponsor**: @pja-ant
- **Extension Identifier**: `io.modelcontextprotocol/skills`
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2640

This SEP was developed by the [Skills Over MCP Working Group](https://modelcontextprotocol.io/community/skills-over-mcp/charter). Design history and experimental findings are maintained in the [ext-skills repository](https://github.com/modelcontextprotocol/ext-skills).

Abstract

This SEP defines a convention for serving Agent Skills over MCP using the existing Resources primitive. A *skill* is a directory of files (minimally a `SKILL.md`) that provides structured workflow instructions to an agent. This extension specifies that each file in a skill directory is exposed as an MCP resource, conventionally under the `skill://` URI scheme. Skills are addressed by URI and may be read directly; a `skills/list` method enumerates the skills a server serves (servers whose skill catalogs are large, generated, or otherwise unenumerable MAY return an empty or partial listing), and a `skills/get` method returns any single skill's entry by URI. The skill format itself (directory structure, YAML frontmatter, naming rules, and the progressive disclosure model that governs how hosts stage content into context) is delegated entirely to the Agent Skills specification; this SEP defines only the transport binding.

The extension defines three protocol methods. Every server declaring the extension implements `skills/list`, which enumerates the skills a server serves, and `skills/get`, which returns the entry for a single skill by URI, including skills absent from the listing. The optional `resources/directory/read` lists the direct children of a directory resource, giving agents scoped navigation of a skill's supporting files. Everything else rides on existing protocol surface, so hosts that already treat MCP resources as a virtual filesystem can consume MCP-served skills identically to local filesystem skills. The specification is accompanied by implementation guidelines for host-provided resource-reading tools and SDK-level convenience wrappers.

Motivation

Native skills support in host applications demonstrates strong demand for rich, progressively disclosed workflow instructions. MCP does not currently offer a conventional way to ship this content alongside the tools it describes, which leads to:

- **Fragmented distribution.** A server and the skill that teaches an agent to use it are versioned, discovered, and installed separately. Users installing a server from a registry have no signal that a companion skill exists. (problem statement)
- **Instruction size limits.** Server instructions are delivered as the `instructions` field of the `server/discover` result and are practically bounded in size. Complex workflows, such as the 875-line mcpGraph skill, do not fit this model. (experimental findings)

- **Inconsistent ad-hoc solutions.** Absent a convention, several independent implementations have each invented their own `skill://` URI structure, with diverging semantics for authority, path, and sub-resource addressing.

Specification

Dependencies

This extension has no dependencies beyond the base MCP Resources primitive. In protocol versions 2026-07-28 and later, `skills/list` results additionally carry the base protocol's list-caching attributes (SEP-2549).

Skill Format

A skill served over MCP MUST conform to the Agent Skills specification. In particular:

- A skill is a directory. Its *skill name* is the value of the `name` field in its `SKILL.md` frontmatter.
- Every skill MUST contain a `SKILL.md` file at its root.
- `SKILL.md` MUST begin with YAML frontmatter containing at minimum the `name` and `description` fields as defined by the Agent Skills specification.
- A skill MAY contain additional files and subdirectories (references, scripts, examples, assets).

This extension does not redefine, constrain, or extend the skill format. Future revisions of the Agent Skills specification apply automatically. In the event that the Agent Skills specification changes in a backwards incompatible way, clients MUST honor any backwards compatibility mechanisms provided by the Agent Skills specification and SHOULD continue to support the Agent Skills specification as it existed prior to any incompatible change.

Resource Mapping

Each file within a skill directory is exposed as an MCP resource. Servers SHOULD use the `skill://` URI scheme, under which the resource URI has the form:

```
skill://<skill-path>/<file-path>
```

where:

- `<skill-path>` is a `/`-separated path of one or more segments locating the skill directory within the server's skill namespace. It MAY be a single segment (`git-workflow`) or nested to arbitrary depth (`acme/billing/refunds`).
- `<file-path>` is the file's path relative to the skill directory root, using `/` as the separator.

The resource for the skill's required `SKILL.md` is therefore always addressable as `skill://<skill-path>/SKILL.md`, and the skill's root directory is `skill://<skill-path>` (the `/SKILL.md` suffix removed, no trailing slash), matching Directory Listing.

The final segment of `<skill-path>` MUST equal the skill's `name` as declared in its `SKILL.md` frontmatter. This mirrors the Agent Skills specification's requirement that `name` match the parent directory name. Preceding segments, if any, are a server-chosen organizational prefix. Servers MAY organize skills hierarchically by domain, team, version, or any other axis. In `skill://acme/billing/refunds/SKILL.md`, the prefix is `acme/billing` and the skill's `name` is `refunds`; in `skill://git-workflow/SKILL.md` there is no prefix and the `name` is `git-workflow`. This means the skill name is always recoverable from the URI alone, without reading frontmatter.

Further constraints:

- A `SKILL.md` MAY appear in a descendant directory of a skill, so skills can nest. See Nested skills.
- The final `<skill-path>` segment, being the skill `name`, MUST satisfy the Agent Skills specification's naming rules. The first `<skill-path>` segment occupies the authority component and SHOULD be a valid `reg-name` per RFC 3986; any other prefix segments SHOULD be valid URI path segments; no further constraints are imposed on them.

Per RFC 3986, the first segment of `<skill-path>` occupies the authority component. This carries no special semantics under this convention and clients MUST NOT attempt DNS or network resolution of it.

A server MAY serve skills under another scheme native to its domain (e.g., `github://owner/repo/skills/refunds/SKILL.md`). No scheme is privileged: the structural constraints above (`<skill-path>` ending in the skill name, `SKILL.md` explicit in the URI) apply regardless of scheme, and `skills/list` enumerates a server's skills whatever scheme they use.

Skill identity does not depend on the scheme. A host learns that a resource is a skill in one of two ways: from a `skills/list` entry, the authoritative record of the skills a server publishes; or from an explicit reference (the server's `instructions` field, another skill, or the user), which `skills/get` confirms, the server answering for a skill it serves and erroring otherwise. This holds for every scheme, `skill://` included. A host MUST NOT conclude that a resource is a skill merely because its URI carries a particular scheme.

Examples

Skill path	File	Resource URI
`git-workflow`	`SKILL.md`	`skill://git-workflow/SKILL.md`
`pdf-processing`	`references/FORMS.md`	`skill://pdf-processing/references/FORMS.md`
`pdf-processing`	`scripts/extract.py`	`skill://pdf-processing/scripts/extract.py`
`acme/billing/refunds`	`SKILL.md`	`skill://acme/billing/refunds/SKILL.md`
`acme/billing/refunds`	`examples/email.md`	`skill://acme/billing/refunds/examples/email.md`

Resource Metadata

For each `skill://<skill-path>/SKILL.md` resource:

- `mimeType` SHOULD be `text/markdown`.
- `name` SHOULD be set from the `name` field of the `SKILL.md` YAML frontmatter. By the path constraint above, this will equal the final segment of `<skill-path>`.
- `description` SHOULD be set from the `description` field of the `SKILL.md` YAML frontmatter.

Servers MAY expose additional frontmatter fields via the resource's `_meta` object. When `_meta` keys are used for skill resources, implementations SHOULD use the `io.modelcontextprotocol.skills/` reverse-domain prefix. Other files in the skill use the `mimeType` appropriate to their content.

Nested skills

A skill directory MAY contain further skills in descendant directories. A nested skill is subject to the same rules as any other skill (its directory name is its `name`, and the enclosing skill's path becomes part of its organizational prefix), with the following semantics:

- **Nested content is supporting content.** From the enclosing skill's perspective, a nested skill's directory and files are ordinary supporting files, and reading them is ordinary reading. A nested `SKILL.md` read this way is ordinary markdown: hosts MUST NOT act on its frontmatter.

- **Activation requires fresh consent.** Approval is per skill: approving a skill approves that skill alone and says nothing about skills nested within it. Activating a nested skill (loading it as a skill in its own right, whether through the host's skill-loading machinery or by giving effect to its frontmatter) requires fresh, explicit user consent; approval of the enclosing skill does not substitute for it. Once activated, a nested skill is an ordinary skill: its frontmatter takes effect under the same rules as any other MCP-served skill, including the approval gate on `allowed-tools`.
- **Publication is flat.** A nested skill is published like any other: through its own `skills/list` entry, or by explicit reference. The listing remains flat: an entry for a nested skill is an ordinary entry whose `uri` happens to share a path prefix with the enclosing skill's, and nothing in the listing marks nesting.

Discovery

A server is not required to make its skills enumerable. A skill's URI is directly readable via `resources/read` whether or not it appears in any listing, and hosts MUST support loading a skill given only its URI (see Hosts: End-to-End Integration). This is the baseline: if a model has the URI, whether from server instructions, another skill, or the user, it can read the skill.

On top of that baseline, three mechanisms are defined. Two are discovery: enumeration via `skills/list`, which every server declaring this extension implements, and an optional pointer from server instructions. The third is retrieval: however a host arrives at a skill's URI, `skills/get` returns that skill's entry (its metadata and digests), including for skills no listing mentions.

Enumeration via `skills/list`

A server declaring the `io.modelcontextprotocol/skills` extension MUST implement the `skills/list` method, which returns the skills it serves. The result MAY be empty.

The request carries an optional pagination cursor:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "skills/list",
  "params": {}
}
```

The result carries the skill entries:

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "resultType": "complete",
    "skills": [
      {
        "uri": "skill://git-workflow/SKILL.md",
        "frontmatter": {
          "name": "git-workflow",
          "description": "Follow this team's Git conventions for branching and commits"
        },
        "resources": [
          {
            "uri": "skill://git-workflow/SKILL.md",
            "digest": "sha256:a1b2c3d4...",
            "size": 2314
          }
        ]
      },
      {
        "uri": "skill://acme/billing/refunds/SKILL.md",
        "frontmatter": {
          "name": "refunds",
          "description": "Process customer refund requests per company policy",
          "license": "Apache-2.0"
        },
        "resources": [
          {
            "uri": "skill://acme/billing/refunds/SKILL.md",
            "digest": "sha256:b2c3d4e5...",
            "size": 3871
          },
          {
            "uri": "skill://acme/billing/refunds/examples/email.md",
            "digest": "sha256:c3d4e5f6...",
            "size": 962
          }
        ]
      },
      {
        "uri": "skill://pdf-processing/SKILL.md",
        "frontmatter": {
          "name": "pdf-processing",
          "description": "Extract, fill, and assemble PDF documents",
          "metadata": { "version": "2.1.0" }
        },
        "resources": [
          {
            "uri": "skill://pdf-processing/SKILL.md",
            "digest": "sha256:d5e6f7a8...",
            "size": 5120
          },
```

```
          {
            "uri": "skill://pdf-processing/references/FORMS.md",
            "digest": "sha256:e6f7a8b9...",
            "size": 18433
          },
          {
            "uri": "skill://pdf-processing/scripts/extract.py",
            "digest": "sha256:f7a8b9c0...",
            "size": 4096
          },
          {
            "uri": "skill://pdf-processing/templates/invoice.md",
            "digest": "sha256:a8b9c0d1...",
            "size": 1210
          },
          {
            "uri": "skill://pdf-processing/templates/purchase-order.md",
            "digest": "sha256:b9c0d1e2...",
            "size": 1388
          },
          {
            "uri": "skill://pdf-processing/templates/regional/eu-invoice.md",
            "digest": "sha256:c0d1e2f3...",
            "size": 1472
          }
        ]
      }
    ]
  }
}
```

Result fields:

Field	Required	Description
`skills`	Yes	Array of skill entries.
`skills[].frontmatter`	Yes	Verbatim copy of the skill's `SKILL.md` YAML frontmatter, rendered as JSON. See Frontmatter.
`skills[].uri`	Yes	Resource URI of the skill's `SKILL.md`. See Skill URIs.
`skills[].resources`	Yes	The skill's files: an array enumerating them with digests and sizes, or the string `"dynamic"`. See Resources.
`skills[].resources[].uri`	Yes	Resource URI of the file.
`skills[].resources[].digest`	Yes	SHA-256 digest of the file. See Integrity.
`skills[].resources[].size`	Yes	Length in bytes of the file's raw content. See Limits.

A skill whose content is generated dynamically carries `"resources": "dynamic"` in place of the array. An entry with no `resources` at all is invalid.

Pagination mirrors the base protocol's list methods: the request accepts an optional `cursor`, and when the result includes `nextCursor` the client passes it back to retrieve the next page. An entry is atomic: a skill's `resources` set is never split across pages.

In protocol versions 2026-07-28 and later, the result also carries the base protocol's list-caching attributes, `ttlMs` and `cacheScope`, as defined for `tools/list` and `resources/list` (SEP-2549), with the same semantics: a freshness hint for the listing and a cache-scope marker, not an integrity property. Integrity and verification governs content regardless of how fresh a cached listing is.

A server whose skill catalog is large, generated on demand, or otherwise unenumerable MAY return an empty or partial listing. Hosts MUST NOT treat an empty or partial listing as proof that a server has no skills. The method serves entries for a server's skills whatever URI scheme they use. Enumeration is uniform across schemes.

Names

A skill's `name` is a label, not an identifier. A skill is identified by its `uri` within a server, and by the pair of server identity and `uri` across servers (Skill URIs). Within a server's listing, names SHOULD be unique, but they are not guaranteed to be: two skills at different paths may share a final segment (`acme/billing/refunds` and `acme/support/refunds` are both named `refunds`), and a nested skill may share its name with a top-level one. Hosts MUST NOT assume name uniqueness. When two entries in one listing collide on `name`, hosts MUST disambiguate them, for example by their distinguishing path segments, rather than silently discarding or preferring one. When skills from different origins collide on `name`, hosts MUST resolve the name within a per-origin namespace, identifying servers by a host-assigned label; an MCP-served skill MUST NOT silently shadow, or be silently substituted for, a same-named skill from any other origin, whether another server's or the host's own filesystem skills. See Security Implications.

Frontmatter

`frontmatter` is the skill's `SKILL.md` YAML frontmatter rendered verbatim as a JSON object. It contains every field the author wrote, not a curated subset. Because the Agent Skills specification requires `name` and `description`, those fields are always present; everything else (`license`, `metadata`, fields added by future revisions of the Agent Skills specification) passes through unchanged. A host can therefore build its skill registry (names, descriptions, and whatever other metadata it understands) from the listing alone, without fetching each `SKILL.md`.

The `frontmatter` object MUST be identical in content to the frontmatter of the `SKILL.md` it describes. The final `<skill-path>` segment of the entry's `uri` MUST equal `frontmatter.name`, per Resource Mapping.

Within the frontmatter `metadata` object, keys prefixed with `io.modelcontextprotocol/` are reserved for metadata defined by MCP extensions. This extension currently defines no such keys. Implementations SHOULD ignore keys under this prefix that they do not recognize.

Skill URIs

`uri` is the full resource URI of the skill's `SKILL.md`, readable via `resources/read`. Supporting files are individually addressable as sibling resources under the same skill path, per Resource Mapping. A skill is always retrieved as individually addressable resources; this extension defines no packed or bundled retrieval form. See Appendix: Deferred Features.

A skill URI is scoped to the server that serves it. Nothing prevents two connected servers from both serving `skill://refunds/SKILL.md`, and those are two unrelated skills. The identity of an MCP-served skill is therefore the pair of the host's identity for the originating server and the skill's `uri`. Hosts MUST preserve both halves wherever a skill is recorded or addressed, including the registry, persisted approvals, the cache, and any tool or path through which the model reaches the skill, and MUST NOT key any of these on the `uri` alone. In particular, any path at which a host materializes skill content, whether a cache directory or a virtual mount, MUST encode the server identity as well as the `uri`, so that same-URI skills from different servers land at distinct paths and

the originating server is recoverable from the path; this is also what lets the host honor the durable-origin requirement in Security Implications.

Resources

`resources` is REQUIRED on every skill entry and takes one of two forms: an array enumerating the skill's files (`SKILL.md` and every supporting file) as `{uri, digest, size}` triples, or the string `"dynamic"`. The array is the unit of content that a host verifies and that a user's approval binds to:

- When present, `resources` MUST be complete: it lists every file of the skill, each exactly once, including an entry matching the skill's top-level `uri`. That entry carries the digest and size of `SKILL.md` itself.
- Each `uri` MUST be the skill's `SKILL.md` or a file within the skill's directory.
- Each entry MUST carry `size`: the length in bytes of the file's raw content (the same bytes the `digest` covers). `size` lets a host budget a skill before fetching anything: it can enforce the Limits from the entry alone, decide whether a file is worth retrieving, and detect a truncated or padded read before hashing it. A read whose byte length differs from the entry's `size` is a verification failure equivalent to a digest mismatch (Integrity and verification), whether or not the host goes on to compute the digest.
- Completeness extends to nested skills: from the enclosing skill's perspective their files are supporting files (Nested skills), so the enclosing skill's `resources` lists them too, and the same file may appear in both the enclosing and the nested skill's entries. A change to a nested skill is therefore a change to the enclosing skill's set.
- When a skill's content is generated dynamically, such that stable digests cannot be published, the server MUST set `"resources": "dynamic"` instead of an array. The marker is explicit so that a host can tell a deliberately unverifiable skill from a malformed entry: an entry with no `resources` at all, or with any value other than an array or `"dynamic"`, is invalid, and hosts MUST NOT load it. A skill whose `resources` is `"dynamic"` offers no content integrity and cannot be content-bound (Security Implications). Hosts MAY decline to load such skills, and server authors SHOULD expect that some hosts will.

Integrity and verification

Digests are SHA-256 hashes of an artifact's raw bytes, formatted as `sha256:{hex}` where `{hex}` is 64 lowercase hexadecimal characters. Each entry in a skill's `resources` carries the digest of the file at its `uri`.

When a host retrieves a file listed in a skill's `resources`, it MUST verify the content against that entry's digest. A mismatch means the content is not what the listing promised. It may be corrupted, tampered with, or simply stale because the skill was updated after the listing was fetched. Whatever the cause, hosts MUST NOT use the unverified content; to recover from staleness, call `skills/get` for that skill (or `skills/list` to refresh the catalog) and proceed from the current `resources` set, which, being different, revokes any content-bound approval (Security Implications). A host is *acting on* a skill from the moment it loads the skill's `SKILL.md` into the model's context until, at the earliest, that `SKILL.md` leaves context; hosts MAY hold the window open longer, never shorter. For the whole of that window the host holds the entry from which it loaded the skill. Because `resources` is complete, an unlisted file is a change to the skill: while acting on a skill, a host MUST resolve reads of the skill's files only to URIs listed in that entry's `resources`, and MUST treat a read of an unlisted file within the skill as a verification failure equivalent to a digest mismatch. Hosts MUST NOT retrieve a skill's files ahead of need, whether on connection, on listing, or at approval. A `SKILL.md` is fetched when the skill is loaded, and a supporting file when it is read. A server may publish many skills with many files each, and every host that connects retrieving all of them would impose load proportional to the catalog rather than to use. Hosts SHOULD instead cache what they do retrieve, and digests make that cache cheap to validate: a cached file whose digest matches the current entry can be served without fetching it again, and one whose digest does not match must be fetched again. A cached copy is only as trustworthy as the host's certainty that its bytes have not changed since they were verified; the requirements on a disk cache are in Security Implications. Lazy retrieval is compatible with content-bound approval, which binds to the entry's `resources` set rather than to retrieved bytes; a file fetched long after approval is verified against that set when it is read.

Digests are unsigned and supplied by the same server that supplies the content. A match proves the two are consistent, not that either is trustworthy. Any intermediary on the path, such as a gateway, can rewrite both the

listing and the content together. Hosts MUST NOT treat a digest match as a security boundary.

After fetching a `SKILL.md` for which the host holds an entry, from either `skills/list` or `skills/get` (digest-verified when the entry's `resources` is an array, and unverifiable when it is `"dynamic"`), hosts MUST parse its YAML frontmatter and compare it field-by-field against the entry's `frontmatter`. Any discrepancy MUST be treated as a verification failure equivalent to a digest mismatch, and the skill MUST NOT be loaded. This enforces the Frontmatter identity requirement on the host side, so that what a user approves from the listing is what the model actually receives.

Limits

This extension fixes two per-skill limits so that servers know what every conforming host will accept and hosts know what they must be prepared to handle:

Limit	Value	Counted over
Resources per skill	512 entries	The entries of the skill's `resources`, `SKILL.md` included
Total file size per skill	16 MiB (16,777,216 bytes)	The sum of `size` over the skill's `resources`

Hosts MUST support skills up to and including these limits, and MAY support larger ones. Servers SHOULD NOT serve a skill that exceeds either limit; a skill that does is not guaranteed to be loadable by any conforming host. Because `resources` is complete, both limits are checkable from the entry alone, by counting entries and summing `size`, before the host retrieves a single file, and a host that declines a skill on this basis SHOULD tell the user why rather than fail silently on a later read.

For a skill whose `resources` is `"dynamic"`, the entry offers nothing to count. A host that chooses to load such a skill applies the total-size limit to what it actually retrieves and MAY stop loading the skill once that limit is reached.

These limits bound a host's exposure to a single skill. They say nothing about how many skills a server may serve or a host must accept; a listing may be arbitrarily large, which is one reason hosts retrieve files only on demand (Integrity and verification).

Retrieval via `skills/get`

A server declaring the `io.modelcontextprotocol/skills` extension MUST also implement the `skills/get` method, which returns the entry for a single skill named by its URI:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "skills/get",
  "params": {
    "uri": "skill://pdf-processing/SKILL.md"
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "resultType": "complete",
    "skill": {
      "uri": "skill://pdf-processing/SKILL.md",
      "frontmatter": {
        "name": "pdf-processing",
        "description": "Extract, fill, and assemble PDF documents",
        "metadata": { "version": "2.1.0" }
      },
      "resources": [
        {
          "uri": "skill://pdf-processing/SKILL.md",
          "digest": "sha256:d5e6f7a8...",
          "size": 5120
        },
        {
          "uri": "skill://pdf-processing/references/FORMS.md",
          "digest": "sha256:e6f7a8b9...",
          "size": 18433
        },
        {
          "uri": "skill://pdf-processing/scripts/extract.py",
          "digest": "sha256:f7a8b9c0...",
          "size": 4096
        },
        {
          "uri": "skill://pdf-processing/templates/invoice.md",
          "digest": "sha256:a8b9c0d1...",
          "size": 1210
        },
        {
          "uri": "skill://pdf-processing/templates/purchase-order.md",
          "digest": "sha256:b9c0d1e2...",
          "size": 1388
        },
        {
          "uri": "skill://pdf-processing/templates/regional/eu-invoice.md",
          "digest": "sha256:c0d1e2f3...",
          "size": 1472
        }
      ]
    }
  }
}
```

`params.uri` MUST be the URI of a skill's `SKILL.md`. The `skill` object is a skill entry, identical in shape and meaning to an entry of `skills/list`, with the same `uri`, `frontmatter`, and `resources` fields under the same rules.

Semantics:

- If the URI does not identify a skill the server serves, the server MUST return error `-32602` (Invalid params), the same code `resources/read` uses for unknown resources.
- A server MUST answer for every skill it serves, whether or not that skill appears in its `skills/list` result. A skill absent from a partial listing is still retrievable by URI.
- The result is a point-in-time snapshot, exactly as a listing entry is. Re-calling the method is how a host refreshes one skill's digests without re-enumerating the catalog.
- A skill whose content is generated dynamically carries `"resources": "dynamic"`, per Resources, whether it is reached through `skills/list` or `skills/get`.
- The result carries no pagination cursor: a single entry is not a list. The entry is a snapshot of the skill as the server holds it at that moment; whether the result should also carry the base protocol's caching attributes (`ttlMs` and `cacheScope`, per SEP-2549), as `resources/read` results do, is left open.

The method complements the baseline: a URI alone is enough to read a skill, and `skills/get` turns that same URI into the skill's metadata and digests, so a skill that never appeared in a listing can still be verified and content-bound (Security Implications).

Pointer from Server Instructions

A server MAY direct the agent to specific skill URIs from its `instructions` field. This requires no discovery machinery on the host; the URI is simply present in the model's context and readable via `resources/read`.

Capability Declaration

Per SEP-2133 extension negotiation, servers declare support for this extension in the `extensions` field of their capabilities:

```
{
  "capabilities": {
    "extensions": {
      "io.modelcontextprotocol/skills": {
        "directoryRead": true
      }
    }
  }
}
```

One extension-specific setting is defined:

Setting	Type	Default	Meaning
`directoryRead`	boolean	`false`	The server implements `resources/directory/read`.

An empty object indicates support for the extension with no optional features. Declaring the extension itself commits the server to `skills/list` and `skills/get`; clients MUST NOT call `resources/directory/read` against a server that has not declared `directoryRead: true`. A server declaring this extension MUST also declare the `resources` capability.

Reading

Skill files are read via the standard `resources/read` method. No skill-specific read semantics are defined.

In particular, reading a `SKILL.md` via `resources/read` does not by itself activate the skill. `resources/read` is transport: it returns bytes, whoever asked for them: a generic resource-reading tool, a resource browser, a user

inspecting the server. A skill is activated only by the host's own skill-loading path, the one that verifies the content against the skill's entry (Integrity and verification), applies any required user approval (Security Implications), and opens the window in which the host is acting on the skill. Hosts MUST NOT treat a `resources/read` of a `SKILL.md` that arrives by any other route as a load: it grants no approval, opens no window, and confers no standing on the skill's supporting files. Content obtained that way is ordinary resource content, and a host that returns it to the model SHOULD do so as it would any other resource read, not as a loaded skill. A host that wishes such a read to load the skill routes it through the skill-loading path instead.

Internal references within a skill (e.g., `SKILL.md` linking to `references/GUIDE.md`) are relative paths, as in the filesystem form of the Agent Skills specification. A client resolves a relative reference against the skill's root, so `references/GUIDE.md` in `skill://acme/billing/refunds/SKILL.md` resolves to `skill://acme/billing/refunds/references/GUIDE.md`, exactly as a filesystem path would resolve. The skill's root is the directory containing `SKILL.md`, not the scheme root. When skills nest, each `SKILL.md`'s references resolve against its own directory: a relative reference in a nested skill's `SKILL.md` resolves against the nested skill's root, regardless of how the file was reached.

Directory Listing

A skill's instructions frequently reference a directory rather than a file: "pick the appropriate template from `templates/`", "run the matching script in `scripts/`". To act on this, the agent must learn what the directory contains. `resources/list` cannot answer that scoped question: it enumerates the server's entire resource space, not a subtree, and the servers this SEP most wants to accommodate, large, generated, or unenumerable catalogs (see Why May the Listing Be Empty or Partial?), may not implement meaningful global listing at all.

This extension therefore defines one new method, `resources/directory/read`, gated behind the `directoryRead` setting of the capability declaration.

Directory resources

A *directory resource* is a resource whose `mimeType` is `inode/directory`. In a skill namespace served as individual files, every directory level is a directory resource: the skill root (`skill://pdf-processing`) and each subdirectory (`skill://pdf-processing/templates`). Directory URIs are written without a trailing slash. Directory resources need not appear in `resources/list`; they are addressable whether listed or not.

`resources/directory/read`

The request carries the directory's URI and an optional pagination cursor. The result carries the resource metadata of the directory's direct children, the same `Resource` objects that `resources/list` returns, with the same `nextCursor` pagination contract.

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "resources/directory/read",
  "params": {
    "uri": "skill://pdf-processing/templates"
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "result": {
    "resultType": "complete",
    "resources": [
      {
        "uri": "skill://pdf-processing/templates/invoice.md",
        "name": "invoice.md",
        "mimeType": "text/markdown"
      },
      {
        "uri": "skill://pdf-processing/templates/purchase-order.md",
        "name": "purchase-order.md",
        "mimeType": "text/markdown"
      },
      {
        "uri": "skill://pdf-processing/templates/regional",
        "name": "regional",
        "mimeType": "inode/directory"
      }
    ]
  }
}
```

Semantics:

- The method applies only to directory resources. If the URI does not exist, or exists but is not a directory resource, the server MUST return error `-32602` (Invalid params), the same code `resources/read` uses for unknown resources.
- The result contains every direct child of the directory: files with their ordinary resource metadata, subdirectories listed as directory resources (`mimeType: "inode/directory"`). The listing is not recursive; clients descend by calling the method again on a child directory.
- An empty directory yields an empty `resources` array.
- Pagination mirrors `resources/list`: when the result includes `nextCursor`, the client passes it back as `cursor` to retrieve the next page.

A server that declares `directoryRead` MUST support the method for every directory within the skill namespaces it serves as individual files. The method itself is not skill-specific: a server MAY support it on any directory resource it serves, under any scheme.

Directory reads and the held entry

For a skill whose entry carries `resources`, the host already holds a complete manifest of the skill's files (Resources); a directory read tells it nothing about that skill's contents that the entry did not. Directory reading earns its place elsewhere: for dynamically generated skills, whose `resources` is `"dynamic"`; for resource trees that are not skills at all; and for obtaining the server's current view of a directory without first refreshing the entry. When a host acting on a skill with a manifest wants to know what `templates/` contains, it MAY answer from the entry alone.

The two views can disagree. If the server adds a file to a skill after the host obtained its entry, a directory read may list that file while the held manifest does not. This is the stale-snapshot case that Integrity and verification already governs, and the recovery path is the one specified there: while acting on the skill under the held entry, the host MUST NOT read the newly listed child (an unlisted file is a verification failure, exactly as a digest

mismatch is) and MUST NOT surface it to the model as a file of the skill. To reach it, the host refreshes the entry with `skills/get`, at which point the `resources` set has changed and any persisted content-bound approval is revoked and must be obtained again (Security Implications). Only under the refreshed entry is the new file readable. Hosts SHOULD expect this sequence and present it as a skill that has changed and needs re-approval, rather than as a read error. Conversely, a child present in the manifest but absent from a directory read is a file the server no longer serves; a read of it will fail, and the same refresh applies.

This extension defines no shared version or cache token that would let a host determine whether a directory result and an entry describe the same snapshot of the server. The manifest is authoritative for what the host may read under its current approval; a directory read is a live observation that may run ahead of or behind it. Hosts MUST NOT treat the directory result as extending the manifest.

For a dynamically generated skill, whose `resources` is `"dynamic"`, none of this changes the skill's standing: it offers no content integrity and cannot be content-bound, and a host MAY decline to load it (Resources). A directory read is how such a skill's files are discovered at all, but it does not supply the integrity the entry lacks.

Implementation Guidelines

The following are recommendations for interoperable implementations. They are not part of the normative specification.

Hosts: End-to-End Integration

This section sketches one way a host might wire MCP-served skills into an existing skills implementation. It is illustrative, not prescriptive. Hosts are free to structure tools, naming, and routing however suits their architecture. The goal is that an MCP-served skill flows through the same loading and reading mechanics as a filesystem skill while remaining origin-tagged, per Security Implications.

Registry. At startup and on connection change, the host assembles a single internal skill registry from every origin it supports: filesystem skill directories, and `skills/list` results from each connected MCP server that declares the `io.modelcontextprotocol/skills` extension. Each registry entry records the skill's `name` and `description` (from the entry's `frontmatter`) and its origin: for a filesystem skill, the local directory, and for an MCP skill, the server identity and the `SKILL.md` resource URI. Assembling the registry reads only the listing: the host MUST NOT fetch `SKILL.md` or any supporting file at this stage (Integrity and verification). The entry's `frontmatter` carries everything the registry needs. The registry is keyed by skill identity, origin and `SKILL.md` URI together (Skill URIs), never by name. Because names collide within and across origins (Names), the registry qualifies colliding names for display rather than dropping either entry; since the name is not the key, a collision never changes how a skill is loaded.

Context. The host surfaces the `name` and `description` of each enabled registry entry in the model's context, the same list the model already sees for filesystem skills, now with MCP-served entries mixed in, together with the entry's identity: the host's label for the originating server and the `SKILL.md` URI. The name and description tell the model what a skill is for; the identity is what the model passes to load it. The host's UI presents the same merged list for user inspection and per-skill enable/disable, with provenance shown so users can see which server a skill came from.

Loading. The host exposes a single skill-loading tool to the model, keyed by skill identity, the originating server and the `SKILL.md` URI:

```
{
  "name": "read_skill",
  "description": "Load a skill's SKILL.md into context.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "server": {
        "type": "string",
        "description": "Name of the connected MCP server"
      },
      "uri": { "type": "string", "description": "The skill's SKILL.md URI" }
    },
    "required": ["server", "uri"]
  }
}
```

`read_skill` is the host's skill-loading path in this sketch, the only route by which a skill is activated (Reading); a `read_resource` call against the same `SKILL.md` URI returns its content but does not load the skill. When the model calls `read_skill`, the host looks up the pair in its registry and routes on origin: a filesystem skill is read from disk; an MCP skill is fetched via `resources/read` against the originating server, at that moment and not before, unless a verified copy is already in the host's cache. The mechanics are the same either way. Keying the tool by identity rather than by name is what lets a URI travel: a skill URI handed to the model by the user, by server instructions, or by another skill's `SKILL.md` is exactly what `read_skill` takes, with no need to map it back to a display name that the host may have qualified to resolve a collision (Names). A host that already exposes a name-keyed loading tool for filesystem skills extends it to accept `server` and `uri` rather than introducing a parallel one; for a filesystem skill it may reserve a `server` value for the local origin and pass the `SKILL.md` path as `uri`.

Supporting files. Once a `SKILL.md` is in context, the model may encounter relative references to supporting files (`references/GUIDE.md`, `scripts/extract.py`). For filesystem skills the model reads these with the host's ordinary file-read tool; for MCP skills there is no local file. The host therefore also exposes a general-purpose resource-reading tool:

```
{
  "name": "read_resource",
  "description": "Read an MCP resource from a connected server.",
  "inputSchema": {
    "type": "object",
    "properties": {
      "server": {
        "type": "string",
        "description": "Name of the connected MCP server"
      },
      "uri": { "type": "string", "description": "The resource URI" }
    },
    "required": ["server", "uri"]
  }
}
```

The host arranges for the model to know, when it loads an MCP-served `SKILL.md`, which server it came from and what its base URI is, for example by stating both in the `read_skill` tool result, so the model can resolve `references/GUIDE.md` to `skill://<skill-path>/references/GUIDE.md` and issue `read_resource` against the right server. A host may instead fold this into its file-read tool by mounting each server's `skill://` namespace into a

virtual path, with one mount root per server so that the path encodes the server identity (Skill URIs), and translating reads under that path into `resources/read` calls, in which case no separate `read_resource` tool is needed and the model treats every supporting file as a local path. A virtual mount resolves reads on access; it MUST NOT be populated by fetching the skill's files in advance. Either way the resolution rule is the same: relative references resolve against the skill's root directory, exactly as on a filesystem. When the skill's entry carries a `resources` array, the host verifies each such read against it, per Integrity and verification.

Directory navigation. Skill instructions may point the model at a directory rather than a file ("choose the right template from `templates/`"). When the originating server declares `directoryRead`, the host SHOULD surface this capability to the model: a `read_resource` call whose target is a directory resource can be routed to `resources/directory/read` and return the child listing, and the virtual-mount approach maps it onto the host's existing directory-listing tool: an `ls` of a mounted path becomes a `resources/directory/read` call.

Unenumerated skills. Because a listing may be empty or partial, a host should also accept skill URIs it has never seen listed, handed to the model by the user, by server instructions, or by another skill. Calling `skills/get` on such a URI yields the same entry a listing would have carried, so an unlisted skill enters the registry, gets verified, and is approved on the same terms as a listed one; a server that does not serve the URI as a skill answers with an error. No special tool surface is needed: `read_skill` already takes a server and a URI, so a `read_skill` call naming a pair the registry has not seen is the trigger for `skills/get`.

Both tool signatures above include `server` because two connected servers may both serve `skill://refunds/SKILL.md` (Skill URIs). That is one disambiguation strategy; a host may instead rewrite URIs with a per-server prefix, scope by session, or anything else appropriate to its architecture. The tool is general-purpose, reads any MCP resource, and is useful beyond skills.

SDKs: Convenience Wrappers

SDK maintainers SHOULD provide affordances that wrap the underlying resource operations in skill-specific terms. For example:

Server-side: declare a skill from a directory, at a given path:

```
@server.skill("git-workflow")                # → skill://git-workflow/SKILL.md
def git_workflow():
    return Path("./skills/git-workflow")

@server.skill("acme/billing/refunds")        # → skill://acme/billing/refunds/SKILL.md
def refunds():
    return Path("./skills/refunds")
```

The SDK handles: reading `SKILL.md` frontmatter to populate resource metadata, serving file content on `resources/read`, and answering `skills/get` (and, where the server's skill set is bounded, `skills/list`), computing entry digests and sizes from the registered files, and warning when a registered skill exceeds the Limits.

Client-side: enumerate and fetch skills:

```
skills = await client.list_skills()              # calls skills/list, paginating; may
be empty
entry = await client.get_skill(
    "skill://acme/billing/refunds/SKILL.md")     # calls skills/get, listed or not
content = await client.read_skill_uri(
    "skill://acme/billing/refunds/SKILL.md")     # wraps resources/read, works
regardless of enumeration
entries = await client.read_directory(
    "skill://pdf-processing/templates")          # wraps resources/directory/read
```

These wrappers are thin, each a single underlying protocol call with a fixed URI pattern, but they give server authors an ergonomic way to declare skills and give client authors a discoverable entry point.

Rationale

The design rationale for this SEP (why skills map to Resources rather than a new primitive, the URI structure, listing semantics, `skills/get`, the choice of a method over an index resource, format delegation to agentskills.io, directory reads, verbatim frontmatter, and per-file digests) is maintained as a standalone document in the Working Group repository: rationale.md.

One point of that design bears restating here because it shapes how the two methods relate. A `skills/list` entry is intentionally a complete manifest of the skill, its verbatim `frontmatter` and its full `resources` set with digests, rather than a summary to be filled in by a follow-up call. A host that pages through the listing therefore has, in that one pass, everything it needs to build its registry, present the skill for approval, bind the approval to content, and verify every file it later reads; there is no second round-trip per skill, which matters most for exactly the hosts that connect to many servers or servers with many skills. `skills/get` exists for the cases the listing does not serve: refreshing a single skill's entry, typically after a digest mismatch, without re-enumerating the catalog, and obtaining an entry for a skill that a partial listing omitted. It is never a step a host must take to complete a listed entry.

Backward Compatibility

This extension introduces three protocol methods. `skills/list` and `skills/get` are implemented by every server declaring the extension, so a client only issues those calls after seeing the declaration, and a client that predates the extension never issues them. `resources/directory/read` is additionally gated behind the `directoryRead` capability setting, so a server that does not declare it never receives the call. The extension introduces no other methods, message types, or schema changes. A server that does not implement this extension simply exposes no `skill://` resources; existing clients are unaffected. A client that does not implement this extension sees `skill://` resources as ordinary resources, which they are.

Existing implementations using other `skill://` URI structures will need to adjust to conform. See the Working Group's related-work survey for a catalog. Notably, FastMCP's widely-used SkillsProvider diverges on URI structure, discovery (per-skill `_manifest` vs. central index), and metadata mapping; coordinating that migration is a near-term Working Group priority. These are mechanical changes, not semantic ones.

Security Implications

Skill content is instructional text delivered to a model, which makes it a prompt-injection surface (background in open-questions.md §10). This extension imposes the following requirements:

- **Skill content is untrusted input.** Hosts MUST treat MCP-served skill content as untrusted model input, subject to the same prompt-injection defenses applied to any server-provided text. A server being

connected does not make its skill content authoritative.

- **Origin MUST be visible to the model.** Hosts MUST tag MCP-served skill content with its originating server identity at the point it enters model context and MUST NOT present an MCP-served skill to the model as indistinguishable from a local filesystem skill. The model, not the host, decides whether to follow a skill's instructions. Withholding origin from it makes the untrusted-input requirement above unenforceable at the layer that acts on it.
- **Skills introduce host-side surfaces that tools do not.** Unlike a remote tool call, an MCP-served skill can place server-authored bytes on the host filesystem and direct the model to execute them with host-side tools. Hosts MUST treat MCP-served skills as a higher-risk surface than remote tool invocation.
- **No implicit local execution.** Hosts MUST NOT allow MCP-served skill content to cause host-side code execution without explicit per-skill user approval. This covers (a) declarative fields the host parses (hooks, frontmatter scripts) and (b) instructions in the skill body that direct the model to invoke any host code-execution tool, whether to run a script bundled in the skill or to run an arbitrary command the skill specifies. Hosts MUST ignore or approval-gate (a), and MUST apply the same approval gate to code-execution tool calls issued while the model is acting on an MCP-served skill.
- **Origin-scoped resource reads.** A model-callable resource-read surface (such as the `read_resource` pattern in Hosts: End-to-End Integration) is a cross-server confused-deputy vector when driven by untrusted skill content. Hosts MUST bind such reads to the skill's originating server: a skill served by server A MUST NOT cause a `resources/read` against server B. Hosts MUST identify servers by a host-assigned label, not the server's self-reported `serverInfo.name`. Any cross-origin read MUST be gated behind explicit per-call user approval naming both servers.
- **Name collisions are an impersonation surface.** Skill names are not unique across origins, and a malicious server can publish a skill under the name of a popular one, whether another server's or the user's own local skill, counting on the host resolving its way. Hosts MUST resolve skill names within a per-origin namespace, identifying servers by a host-assigned label, not the self-reported `serverInfo.name`; MUST NOT let an MCP-served skill silently shadow, replace, or intercept invocations of a same-named skill from any other origin, including the host's filesystem skills; and SHOULD surface collisions to the user. A name binds to whatever bytes its origin currently serves and carries no authorship or endorsement. Intermediaries MAY attach provenance or verification annotations via `_meta` under their own reverse-domain prefix, not the `io.modelcontextprotocol.skills/` prefix reserved for this extension (Resource Metadata); this extension assigns such annotations no semantics.
- **No implicit permission grants.** Hosts MUST NOT honor frontmatter fields that widen the model's tool or filesystem permissions when the skill arrives over MCP. In particular, the Agent Skills `allowed-tools` field, which a filesystem-sourced skill uses to declare the tools available while it runs, MUST be ignored for MCP-origin skills unless the user has explicitly approved that grant for that skill. A remote server populating `allowed-tools` is requesting elevated access on the host, not declaring a property of its own environment. Approval of a skill never extends to the frontmatter of any other `SKILL.md` within its file space: a nested skill's `allowed-tools` has no effect unless that nested skill is itself activated under its own approval (Nested skills).
- **Skills are data, not directives.** Hosts MUST NOT treat skill resources as higher-authority than other context. Explicit user policy governs whether a skill is loaded at all.
- **Nested skill consent.** Approval is per skill: approving a skill does not approve skills nested within it. Activating a nested `SKILL.md` requires fresh, explicit user consent, per Nested skills. Silently promoting a file of an approved skill to an active skill would let a server ride new instructions and permission requests in on a prior approval.
- **Provenance and inspection.** Hosts SHOULD indicate which server a skill originates from when presenting it, SHOULD let users inspect a skill's content before it is loaded into model context, and MAY gate loading behind per-skill or per-server user approval.
- **Digests are not a security boundary.** Listing digests are unsigned and come from the same server as the content. They confirm consistency between the listing and what was fetched, as described in Integrity and verification, but they cannot establish trust in the content, defend against the server itself, or detect an intermediary that rewrites both together.
- **Content-bound approval.** When a host persists any per-skill user approval, it MUST be bound to the entry's `resources` set (every `uri` and `digest`) observed at the moment of approval. If a subsequent entry for that skill, from `skills/list` or `skills/get`, advertises a different set, whether a file was rotated,

added, or removed, the host MUST treat the prior approval as revoked and re-prompt before loading or executing. A host need not poll for changes. While it is acting on the skill (Integrity and verification), content that has moved fails verification when read; and if it does fetch a fresh entry, the rule above revokes the approval. Neither path lets moved content through under the old approval. A skill whose `resources` is `"dynamic"` cannot be content-bound: hosts MAY decline to load it, and MUST NOT treat a persisted approval as covering whatever content the server currently serves. Digest verification (Integrity and verification) defends the approval after it is granted. It cannot establish that the content was trustworthy when the user approved it, because the server authors both the listing and the body.

- **Caching, cache integrity, cache isolation, and durable origin.** Hosts SHOULD cache verified skill content locally, populated on demand as files are read rather than in bulk (Integrity and verification). A cache is a second copy of content that was verified once; the verification does not carry over to bytes that may have changed since. Hosts that cache skill content on disk MUST therefore do one of the following for every file served from the cache: keep the cache where nothing but the host can write to it (not the model, not scripts or tools the model runs, not other users of the machine) and never modify a cached file in place; or recompute the file's SHA-256 digest from the cached bytes on each access and compare it against the entry's digest, treating a mismatch exactly as a mismatch on a fresh read. Comparing a stored digest label, or a modification time, is not verification. Hosts that cache MCP-served skill content on disk MUST also do so in a location excluded from every filesystem-skill discovery path, and MUST treat content loaded from that location as having arrived over MCP for all purposes of the no-implicit-local-execution rule above, including after host restart and after the originating server is disconnected. Cached bytes do not graduate to filesystem-skill trust by residing locally. Hosts SHOULD remove a server's cached skill content when the user removes that server.

Reference Implementation

Per SEP-2133, an Extensions Track SEP requires at least one reference implementation in an official SDK prior to review.

SDK implementations:

- Python SDK: python-sdk#3485
- C# SDK: csharp-sdk#1856
- Go SDK: go-sdk#1238

Conformance tests:

- Client and server scenarios with the SEP traceability file (`sep-2640.yaml`): conformance#330

Prototype host implementations (reading `skill://` resources, surfacing skills alongside filesystem skills):

- gemini-cli: olaservo/gemini-cli#1
- fast-agent: evalstate/fast-agent#815
- codex: olaservo/codex#1
- Claude Code: prototyped internally at Anthropic; not yet public

Prototype server implementation:

- GitHub MCP Server: github/github-mcp-server#3046

Appendix: Deferred Features

Features recorded here appeared in earlier revisions of this SEP and were removed before review concluded. They are not part of this extension. Each is kept on record with the objections that removed it, so that any future proposal to reintroduce one starts from those objections rather than rediscovering them.

Archive Distribution

An earlier revision let a skill entry advertise pre-packed archives of the entire skill directory (gzip-compressed tar and ZIP) as an alternative retrieval form alongside the skill's `uri`. A host could fetch a multi-file skill in a single `resources/read` rather than one per file, and an archive could carry UNIX file metadata (executable bits, symlinks) that individually served resources cannot represent.

The Core Maintainers removed archives during review, for two reasons:

- **Unpacking is an attack surface disproportionate to the benefit.** Safely extracting an archive supplied by a remote server means defending against decompression bombs, path traversal, links resolving outside the skill directory, case- and Unicode-normalization collisions that silently overwrite `SKILL.md`, setuid and setgid bits, and non-regular file entries such as device nodes. Every host would have to implement that checklist correctly, and a host that got any item wrong would be exploitable by any server it connects to. Serving a skill as individually addressable resources has no comparable surface.
- **Two ways to serve one skill is a compatibility hazard.** Archives were a second encoding of content the protocol could already express. Hosts would have to support both forms to be certain of reading any skill, and a skill offered only as an archive would be unreadable to a host that implemented individual-file reads alone. A single retrieval form keeps the compatibility floor flat: any conforming host can read any conforming skill.

The cost of removal is the one archives were introduced to address: a skill with many supporting files takes one round trip per file, and executable bits and symlinks have no representation. Because hosts retrieve files only as they are needed (Integrity and verification), that cost scales with the files a session actually uses rather than with the size of the skill. Should archives be reconsidered, the questions to settle first are how to bound host-side unpacking risk, perhaps by restricting the format to a profile admitting no symlinks, no non-regular entries, and a declared uncompressed size; and how to keep an archive strictly an optimization, never the sole way to retrieve a skill, so that the compatibility floor stays flat. An archive form would now also be required to unpack to exactly the file set enumerated in the entry's `resources` (Resources).

References

- Agent Skills specification
-
- SEP-2549: TTL for list results
- SEP-2076: Agent Skills as first-class primitive (alternative approach)
- Skills Over MCP Working Group charter
- Decision Log: Working Group decisions and rationale
- Experimental Findings: results from implementations (WIP)
- Related Work: survey of existing skill-serving implementations
- Skill `_meta` Keys: `_meta` key conventions for skill resources
- RFC 3986: URIs

SEP-2663 Tasks Extension

Final · Extensions Track · Created 2026-04-27

- **Status**: Final
- **Type**: Extensions Track
- **Created**: 2026-04-27
- **Author(s)**: Luca Chang (@LucaButBoring), Caitie McCaffrey (@CaitieM20); on behalf of the Agents Working Group
- **Sponsor**: Caitie McCaffrey (@CaitieM20)
- **Extension Identifier**: `io.modelcontextprotocol/tasks`
- **PR**: https://github.com/modelcontextprotocol/modelcontextprotocol/pull/2663

Abstract

This SEP defines an extension that allows a server to respond to a `tools/call` request with an asynchronous *task handle* instead of a final result, allowing the client to retrieve the eventual result by polling. The extension introduces three methods: `tasks/get`, `tasks/update`, and `tasks/cancel`; a polymorphic-result discriminator (`resultType: "task"`); and a `Task` shape that carries a task status, in-progress server-to-client requests, and a final result or error. Task creation is server-directed: the client signals support by including the extension in its per-request capabilities, and the server decides on a per-request basis whether to materialize a task.

Tasks will become a foundational building block of MCP and are expected to be supported in future protocol versions. The experimental `tasks` feature in the `2025-11-25` specification served as a stopgap until the protocol's extension mechanism was available. Now that extensions have been formalized, moving tasks to an official extension gives the feature time to incubate and evolve based on additional real-world implementation feedback, without being constrained by the core specification's release cadence. Once the extension has stabilized and achieved broad adoption, it is intended to be promoted into the core protocol.

This proposal *removes* the version of tasks specified in the `2025-11-25` release from the core protocol and moves it to an Extension. It also proposes updates to Tasks shaped by implementation feedback since that release, and by several changes to the base protocol included in the `2026-06-30` specification:

- SEP-2260: Require Server requests to be associated with a Client request
- SEP-2322: Multi Round-Trip Requests
- SEP-2243: HTTP Header Standardization for Streamable HTTP Transport
- SEP-2567: Sessionless MCP via Explicit State Handles
- SEP-2575: Make MCP Stateless

Motivation

The experimental tasks feature served as an alternate execution mode for tool calls, elicitation, and sampling, allowing receivers to return a poll handle instead of blocking until a final result was ready. Implementation experience surfaced several challenges:

1. **The handshake is fragile.** Tasks today expose method-level capabilities (`tasks.requests.tools.call` declares that `tools/call` **MAY** be task-augmented) alongside a tool-level `execution.taskSupport` field that declares whether a particular tool will accept the augmentation. Clients express their own support for tasks by passing a `task` parameter on their requests, but **MUST NOT** include it if the method/tool does not

support tasks. A client that wants to opt into tasks must therefore prime its state with a `tools/list` call before issuing any task-augmented request, and cannot blindly attach a `task` parameter to every request to handle tools isomorphically. This is confusing, implicit, and easy to get wrong.

2. **`tasks/result` is a blocking trap.** In the current flow, a client that observes `input_required` is required to call `tasks/result` prematurely so that the server has an SSE stream on which to side-channel elicitation or sampling requests. `tasks/result` then blocks until the entire operation completes. This forces long-lived persistent connections that many clients and servers do not want to implement, and it conflicts with SEP-2260, which disallows unsolicited server-to-client requests outright. Under SEP-2260, the SSE semantics that justified the blocking behavior no longer apply.

3. **`tasks/list` scoping cannot be defined.** To avoid clients cancelling or retrieving results for tasks they shouldn't have access to, all tasks should be bound to some sort of "authorization context," the implementation of which is left to individual servers according to their existing bespoke permission models. However, in many cases, it is not possible to perform this binding, in which case the task ID becomes the only line of defense against contamination. In this scenario, it is unsafe for a server to support `tasks/list` at all. While it was possible for tasks to instead be bound to a session, SEP-2567 removes sessions from the protocol. There is no other natural scope a server can define unilaterally — task IDs can be unguessable handles that a server can recognize one at a time, but servers cannot reliably correlate two unrelated handles to the same caller without additional state.

Beyond implementation challenges, tasks face another structural issue: **Client-hosted tasks are no longer expressible.** SEP-1686 permitted clients to host tasks for elicitation and sampling, in part to avoid coupling tasks to tool calls. SEP-2260 makes any unsolicited server-to-client request invalid; every server-to-client polling request under client-hosted tasks would be unsolicited by definition.

This proposal intends to solve the above issues by redesigning certain aspects of the feature and moving tasks out to an official extension. Redefining tasks as an official extension gives the feature more time to incubate and evolve independently of the core specification, promoting adoption. As part of the redesign, this proposal consolidates the polling lifecycle into `tasks/get` and a new `tasks/update` to remove the blocking `tasks/result` method. The redesign allows servers to return tasks unsolicited (in response to ordinary, non-`task`-flagged requests) to eliminate the per-request opt-in and the `tools/list` warmup, relying instead on the extension capability as the single handshake point. Finally, this proposal removes client-hosted elicitation and sampling tasks in compliance with SEP-2260.

Specification

The MCP Tasks extension allows certain requests to be augmented with **tasks**. Tasks are durable state machines that carry information about the underlying execution state of the request they augment, and are intended for client polling and deferred result retrieval. Each task is uniquely identifiable by a server-generated **task ID**.

Tasks are useful for representing expensive computations and batch processing requests, and map naturally onto external job APIs.

Extension Identifier

This extension is identified as: `io.modelcontextprotocol/tasks`.

Capability Negotiation

The client and server declare support for the tasks extension in their respective capabilities objects (using updated form from SEP-2575: Make MCP Stateless):

```
// Client to server, in per-request capabilities
{
  // Other request parameters...
  "params": {
    "_meta": {
      "io.modelcontextprotocol/clientCapabilities": {
        "extensions": {
          "io.modelcontextprotocol/tasks": {},
        },
      },
    },
  },
}
```

```
// Server to client, in response to server/discover
{
  "result": {
    // Other response parameters...
    "capabilities": {
      "extensions": {
        "io.modelcontextprotocol/tasks": {},
      },
    },
  },
}
```

No extension-specific settings are currently defined; an empty object indicates support.

A server that has negotiated this extension **MAY** return `CreateTaskResult` in lieu of a standard result (e.g. `CallToolResult`) in response to any supported request at its own discretion and on a per-request basis. The server is the sole decider; clients do not signal task preference on the request itself. The client declaring the extension capability does not suggest that it requires a `CreateTaskResult` in response to that request.

A server **MUST NOT** return `CreateTaskResult` to a client that did not include the extension capability on its request, regardless of prior declarations. A client that has negotiated this extension **MUST** be prepared to handle either `CallToolResult` or `CreateTaskResult` in response to any supported request it issues. A client that receives `CreateTaskResult` in response to an unsupported request type **MUST** interpret this as an invalid response to the request.

If a server is unable to service a request to a client that does not declare this extension capability without returning `CreateTaskResult`, the server **MUST** return an error with the code `-32021` (Missing Required Client Capability), indicating the required extension in the error response:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "error": {
    // MISSING_REQUIRED_CLIENT_CAPABILITY
    "code": -32021,
    // Message provided for example purposes only. The content of this example message
is non-normative.
    "message": "Missing required client capability",
    "data": {
      "requiredCapabilities": {
        "extensions": {
          "io.modelcontextprotocol/tasks": {}
        }
      }
    }
  }
}
```

Supported Methods

The following methods currently support task-augmented execution:

- `tools/call`

This specification may be extended to support tasks over other request types in the future; implementations **SHOULD** be designed to accommodate additional request types in future revisions of this specification.

Polymorphic Results

A request that is eligible for task-augmentation may return one of two distinct result shapes — the request's standard result, or a `CreateTaskResult`. The discriminator is the `resultType` field on the result object, introduced by SEP-2322:

```
// "task" is introduced by this extension.
type ResultType = "complete" | "input_required" | "task" | string;
```

Servers **MUST** set `resultType` to `"task"` when returning a `CreateTaskResult` so that clients can distinguish it from a standard result. Servers **MUST NOT** set `resultType` to `"task"` on result types other than `CreateTaskResult`.

Client implementors are advised that existing code returning a fixed shape (e.g., a `tools/call` method returning `CallToolResult`) need not change their public contract — they can transparently drive the polling flow internally and surface only the final, completed result. New implementation surfaces **MAY** expose the task lifecycle directly for applications able to leverage it.

Tasks

A `Task` carries operational metadata about ongoing work.

```ts
interface Task {
  /** Stable identifier for this task. */
  taskId: string;

  /** Current task status. */
  status: "working" | "input_required" | "completed" | "cancelled" | "failed";

  /**
   * Optional message describing the current task state.
   * This can provide context for any status, for example (non-normative):
   * - Progress descriptions for "working"
   * - Work blocked on "input_required"
   * - Reasons for "cancelled" status
   * - Summaries for "completed" status
   * - Additional information for "failed" status (e.g., error details, what went wrong)
   *
   * This MAY be exposed to the end-user or model.
   */
  statusMessage?: string;

  /** ISO 8601 timestamp when the task was created. */
  createdAt: string;

  /** ISO 8601 timestamp when the task was last updated. */
  lastUpdatedAt: string;

  /**
   * Time-to-live duration from creation in integer milliseconds, null for unlimited.
   * The server may discard the task after the TTL elapses. This value MAY change
   * over the lifetime of a task.
   */
  ttlMs: number | null;

  /**
   * Suggested polling interval in integer milliseconds. Clients SHOULD honor
   * this value to avoid overwhelming the server. This value MAY change over
   * the lifetime of a task.
   */
  pollIntervalMs?: number;
}
```

Task Status

Tasks can be in one of the following states:

- `working`: The request is currently being processed.
- `input_required`: The server needs input from the client before the task can proceed. The `tasks/get` response will include outstanding requests in the `inputRequests` field. The client **MUST** inspect this field and **SHOULD** provide responses via the `inputResponses` field in subsequent `tasks/update` requests.
- `completed`: The request completed successfully and results are available in the `result` field. This includes tool calls that returned results with `isError: true`.
- `failed`: The request failed due to a JSON-RPC error during execution. The task will include the `error` field with the JSON-RPC error details. This status **MUST NOT** be used for non-JSON-RPC errors.

- `cancelled`: The request was cancelled before completion.

Derived shapes of `Task` inline status-specific payload fields and are used by `tasks/get` responses and `notifications/tasks` notifications:

```ts
/**
 * A task that is in a normal working state.
 * Used by tasks/get and notifications/tasks.
 */
export interface WorkingTask extends Task {
  status: "working";
}

/**
 * A task that is waiting for input from the client.
 * Used by tasks/get and notifications/tasks.
 */
export interface InputRequiredTask extends Task {
  status: "input_required";
  /**
   * Server-to-client requests that need to be fulfilled during task execution.
   * Keys are arbitrary identifiers for matching requests to responses.
   */
  inputRequests: InputRequests;
}

/**
 * A task that has completed successfully.
 * Used by tasks/get and notifications/tasks.
 */
export interface CompletedTask extends Task {
  status: "completed";
  /**
   * The final result of the task.
   * The structure matches the result type of the original request.
   * For example, a CallToolRequest task would return the CallToolResult structure.
   */
  result: JSONObject;
}

/**
 * A task that has failed due to a JSON-RPC error.
 * Used by tasks/get and notifications/tasks.
 */
export interface FailedTask extends Task {
  status: "failed";
  /**
   * The JSON-RPC error that caused the task to fail.
   */
  error: JSONObject;
}

/**
 * A task that has been cancelled.
 * Used by tasks/get and notifications/tasks.
 */
export interface CancelledTask extends Task {
  status: "cancelled";
```

```
}

/**
 * A union type representing a task with optional inlined result/error/inputRequests
fields.
 * This type is used by tasks/get and notifications/tasks to provide complete task state
 * including terminal results or pending input requests.
 */
export type DetailedTask =
  WorkingTask | InputRequiredTask | CompletedTask | FailedTask | CancelledTask;
```

Task Creation

A server returns `CreateTaskResult` in lieu of the standard result shape for a request to indicate that request will be processed asynchronously.

```
// resultType: "task"
type CreateTaskResult = Result & Task;
```

Example Request (CallToolRequest):

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "tools/call",
  "params": {
    "name": "get_weather",
    "arguments": {
      "city": "New York"
    }
  }
}
```

Example Response (CreateTaskResult):

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "result": {
    "resultType": "task",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "working",
    "statusMessage": "The operation is now in progress.",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:40:00Z",
    "ttlMs": 60000,
    "pollIntervalMs": 5000
  }
}
```

The embedded `task` is the seed state for the task, typically (though not necessarily) with `status: "working"`. The client uses `task.taskId` for all subsequent `tasks/get`, `tasks/update`, and `tasks/cancel` calls.

A server **MUST NOT** return `CreateTaskResult` until the task is durably created — that is, until a `tasks/get` for the returned `taskId` would resolve. In eventually-consistent environments, the server **MUST** wait for consistency before responding. This requirement eliminates the need for clients to speculatively poll for task creation.

Server implementations that use multi round-trip requests in conjunction with task creation (for example, a tool that requires elicitation over `InputRequiredResult` before creating a task) **SHOULD** resolve all MRTR exchanges *synchronously* before responding with a `CreateTaskResult`.

Task Polling

Clients poll for task completion by sending `tasks/get` requests.

Clients **SHOULD** respect the `pollIntervalMs` provided in responses when determining polling frequency. The `pollIntervalMs` **MAY** change over the lifetime of a task. Servers **MAY** rate-limit clients polling more frequently than the recorded `pollIntervalMs`.

Clients **SHOULD** continue polling until the task reaches a terminal status or until invoking `tasks/cancel`. Clients **SHOULD** persist task IDs to durable storage so that polling can resume after a crash or restart.

Request

```
interface GetTaskRequest extends JSONRPCRequest {
  method: "tasks/get";
  params: {
    /** Identifier of the task to query. */
    taskId: string;
  };
}
```

Response

Upon receiving a `tasks/get` request, the server **MUST** check the status of the task and respond accordingly:

1. If the status is `working`, the server **MUST** return a a `Task` object with status `working`.
2. If the status is `input_required`, the server **MUST** return a `Task` object with status `input_required` and an `inputRequests` field defined in Multi Round-Trip Requests. The `inputRequests` field **MUST** contain all outstanding requests from the server to the client that need to be fulfilled before the task can proceed.
3. If the status is `completed`, the server **MUST** return a `Task` object with status `completed` and a `result` field containing the final result of the task.
4. If the status is `cancelled`, the server **MUST** return a `Task` object with status `cancelled`.
5. If the status is `failed`, the server **MUST** return a `Task` object with status `failed` and the error that occurred during execution.

```
type GetTaskResult = Result & DetailedTask;
```

The response carries the appropriate response variant for the task's current status (see Task Status). The `resultType` field **MUST** be set to `"complete"` on this object as it is the standard result shape for the `tasks/get` request.

If the task has a non-null `ttlMs`, clients **MAY** treat the TTL as a backstop: if the task's observable status has not reflected the update after `createdAt` plus `ttlMs` has elapsed, the client **MAY** consider the task to no longer be

usable. Conversely, servers **MAY** mark a task as `failed` at any point after the TTL elapses, and subsequently delete it at any time. The value of `ttlMs` **MAY** change over the lifetime of a task.

Task Update Requests

When a task requires input from the client (indicated by the `input_required` status), the server includes outstanding requests in the `inputRequests` field of the `tasks/get` response (see Multi Round-Trip Requests). The client provides responses via the `inputResponses` field in one or more subsequent `tasks/update` requests.

When a client observes a `tasks/get` response (or `notifications/tasks` notification) with `status: "input_required"`, the client **SHOULD** fulfill the outstanding requests in `inputRequests` by sending one or more `tasks/update` requests with corresponding `inputResponses`. After sending a `tasks/update`, the client **SHOULD** continue observing the task's status via polling (`tasks/get`) or notifications (`notifications/tasks`) until it reaches a terminal state.

Clients **MUST** treat each entry in `inputRequests` as they would the equivalent standalone server-to-client request — for example, an elicitation request surfaced via `inputRequests` is subject to the same trust model and user-facing behavior as a direct `elicitation/create` request. Clients **SHOULD** deduplicate `inputRequests` keys across consecutive polls to avoid presenting the same request to the user or model more than once.

Each request key in `inputRequests` **MUST** be unique over the lifetime of a single task. A server **MUST NOT** reuse a key for a subsequent server-to-client request after a response for that key has been delivered, and **MUST NOT** use the same key to refer to two distinct requests over a task's lifetime. This guarantees that `inputResponses` keyed by the same identifier always refer to the request the client expects, eliminates ambiguity for clients deduplicating across polls, and lets servers ignore `inputResponses` for unknown or already-satisfied requests.

Request

```
interface UpdateTaskRequest extends JSONRPCRequest {
  method: "tasks/update";
  params: {
    /** Identifier of the task to update. */
    taskId: string;

    /**
     * Responses to outstanding inputRequests previously surfaced by the
     * server. Shape per MRTR. Each key MUST correspond to a currently-
     * outstanding inputRequest key.
     */
    inputResponses: InputResponses;
  };
}
```

Response

```
type UpdateTaskResult = Result; // empty acknowledgement
```

On success, the server **MUST** acknowledge the request with an empty result. The acknowledgement is *eventually consistent*: the server **MAY** accept the responses and return the ack before the task's observable status (via `tasks/get` or `notifications/tasks`) reflects them. Servers **SHOULD** return a JSON-RPC error if the

`taskId` does not correspond to a known task. Clients **SHOULD** track `inputRequests` keys to avoid responding to requests more than once.

A server **SHOULD** ignore any `inputResponses` responses mapped to a key that is not currently outstanding for the task — including keys that were never issued, keys that have already been answered, and keys whose corresponding request has been superseded. A server **MAY** accept a partial set of responses (a strict subset of currently-outstanding keys);

The `resultType` field **MUST** be set to `"complete"` on `UpdateTaskResult` as it is the standard result shape for the `tasks/update` request.

Task Cancellation

A client sends a `tasks/cancel` request to signal its intent to cancel an in-progress task. The `notifications/cancelled` notification **MUST NOT** be used for task cancellation.

Request

```
interface CancelTaskRequest extends JSONRPCRequest {
  method: "tasks/cancel";
  params: {
    taskId: string;
  };
}
```

Response

```
type CancelTaskResult = Result; // empty acknowledgement
```

The server **MUST** acknowledge the request with an empty result. Servers **SHOULD** return a JSON-RPC error if the `taskId` does not correspond to a known task. Cancellation processing is *eventually consistent* — the task's observable status **MAY** remain `working` (or some other non-terminal status) after the ack, and **MAY** ultimately reach a terminal status other than `cancelled` if the work finished before cancellation could take effect.

Cancellation is **cooperative**: The request signals intent, and the server decides whether and when to honor it. A server is not obligated to actually stop the work; it is only obligated to acknowledge the request. Eventual transition to `cancelled` is not guaranteed.

Clients **MAY** delete all state associated with the task as soon as they send a cancellation (e.g., it no longer needs to retain the list of `inputRequests` keys that it has already responded to). The client does not need to poll `tasks/get` again to wait for the task to reach the `cancelled` state.

The `resultType` field **MUST** be set to `"complete"` on `CancelTaskResult` as it is the standard result shape for the `tasks/cancel` request.

Task Status Notifications

Servers **MAY** push status updates via `notifications/tasks` notifications in addition to servicing client polls:

```ts
export type TaskStatusNotificationParams = NotificationParams & Task;

export interface TaskStatusNotification extends JSONRPCNotification {
  method: "notifications/tasks";
  params: TaskStatusNotificationParams;
}
```

To begin listening for task status notifications, clients send a `subscriptions/listen` request to the server including a list of task IDs the client is interested in (see SEP-2575):

```ts
export interface SubscriptionsListenRequest extends Request {
  method: "subscriptions/listen";
  params: {
    // Other existing fields...
    notifications: {
      taskIds?: string[];
      // Other existing fields...
    };
  };
}
```

In its acknowledgement notification, the server includes the list of task IDs it has agreed to send task status notifications for, if any:

```ts
export interface SubscriptionsAcknowledgedNotification extends Notification {
  method: "notifications/subscriptions/acknowledged";
  params: {
    notifications: {
      /**
       * Subscribe to notifications/tasks for specific task IDs.
       */
      taskIds?: string[];
      // Other existing fields...
    };
  };
}
```

If a client requests task status notifications but does not declare the `io.modelcontextprotocol/tasks` extension capability, the server **MUST** return a JSON-RPC error specifying the missing capabilities:

```
{
  "jsonrpc": "2.0",
  "id": 12,
  "error": {
    // MISSING_REQUIRED_CLIENT_CAPABILITY
    "code": -32021,
    // Message provided for example purposes only. The content of this example message
is non-normative.
    "message": "Missing required client capability",
    "data": {
      "requiredCapabilities": {
        "extensions": {
          "io.modelcontextprotocol/tasks": {}
        }
      }
    }
  }
}
```

Each notification carries a complete `DetailedTask` for the current status, identical to what `tasks/get` would have returned at that moment.

Notification:

```
{
  "jsonrpc": "2.0",
  "method": "notifications/tasks",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "completed",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 60000,
    "pollIntervalMs": 5000,
    "result": {
      "content": [
        {
          "type": "text",
          "text": "Operation completed successfully."
        }
      ],
      "isError": false
    }
  }
}
```

The notification includes the full task object, allowing clients to access the complete task state and final results without polling the `tasks/get` method. Clients **MAY** continue polling `tasks/get` in addition to subscribing to task status notifications, but need not do so.

`notifications/progress` and `notifications/message` notifications **MUST NOT** be sent on the `subscriptions/listen` stream for a task, and are not supported on tasks in general in this specification.

Streamable HTTP: Routing Headers

When `tasks/get`, `tasks/update`, or `tasks/cancel` is sent over the Streamable HTTP transport, the client **MUST** set the `Mcp-Name` header (defined by SEP-2243) to the value of `params.taskId`. This allows transport intermediaries and load balancers to route subsequent requests for the same task to the server instance holding its state, which is typically required for correctness. The `Mcp-Method` header is set to the JSON-RPC method name per SEP-2243.

Example Message Flow

Consider a simple tool call, `hello_world`, requiring an elicitation for the user to provide their name. The tool itself takes no arguments.

To invoke this tool, the client makes a `CallToolRequest` as follows:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "tools/call",
  "params": {
    "name": "hello_world",
    "arguments": {},
    "_meta": {
      // Other metadata...
      "io.modelcontextprotocol/clientCapabilities": {
        "extensions": {
          "io.modelcontextprotocol/tasks": {},
        },
      },
    },
  },
}
```

The server determines (via bespoke logic) that it wants to create a task to represent this work, and it immediately returns a `CreateTaskResult`:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "result": {
    "resultType": "task",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "working",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 3600000,
    "pollIntervalMs": 5000
  }
}
```

Once the client receives the `CreateTaskResult`, it begins polling `tasks/get`:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

On each request while the task is in a `"working"` status, the server returns a regular task response:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "resultType": "complete",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "working",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 3600000,
    "pollIntervalMs": 5000
  }
}
```

Eventually, the server reaches the point at which it needs to send an elicitation to the user. It sets the task status to `"input_required"` to signal this. On the next `tasks/get` request from the client, the server sends the elicitation payload via the `inputRequests` field. Note that while task `inputRequests` share structural similarities with SEP-2322 multi round-trip requests, they are a distinct mechanism: task `inputRequests` are surfaced via `tasks/get` and fulfilled via `tasks/update`, not via retries of the original method. A server that needs client input *before* returning a `CreateTaskResult` (e.g. to decide whether to proceed) uses the multi round-trip request flow on the original request; a server that needs client input *during* task execution uses the `inputRequests` / `inputResponses` mechanism described here.

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

```
{
  "id": 4,
  "jsonrpc": "2.0",
  "result": {
    "resultType": "complete",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "input_required",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 3600000,
    "pollIntervalMs": 5000,
    "inputRequests": {
      "name": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please enter your name.",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "name": { "type": "string" }
            },
            "required": ["name"]
          }
        }
      }
    }
  }
}
```

For thoroughness, let's consider a case where the client happens to poll `tasks/get` again *before* the user has fulfilled the elicitation request. As `inputRequests` is effectively a point-in-time snapshot of all outstanding server-to-client requests associated with the task, the server includes the same request again, despite the client having already seen this information (the client is advised to deduplicate `inputRequests` with the same key for UX purposes):

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

```
{
  "id": 5,
  "jsonrpc": "2.0",
  "result": {
    "resultType": "complete",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "input_required",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 3600000,
    "pollIntervalMs": 5000,
    "inputRequests": {
      "name": {
        "method": "elicitation/create",
        "params": {
          "mode": "form",
          "message": "Please enter your name.",
          "requestedSchema": {
            "type": "object",
            "properties": {
              "name": { "type": "string" }
            },
            "required": ["name"]
          }
        }
      }
    }
  }
}
```

The user enters their name, and the client makes a `tasks/update` request with the satisfied information:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "method": "tasks/update",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "inputResponses": {
      "name": {
        "action": "accept",
        "content": {
          "input": "Luca"
        }
      }
    }
  }
}
```

The server acknowledges the request:

```
{
  "jsonrpc": "2.0",
  "id": 6,
  "result": {
    "resultType": "complete"
  }
}
```

Asynchronously, the server processes it and moves the task back into the `working` status:

```
{
  "jsonrpc": "2.0",
  "id": 7,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

```
{
  "id": 7,
  "jsonrpc": "2.0",
  "result": {
    "resultType": "complete",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "working",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 3600000,
    "pollIntervalMs": 5000
  }
}
```

Eventually, the server completes the request, so it stores the final `CallToolResult` and moves the task into the `"completed"` status. On the next `tasks/get` request, the server sends the final tool result inlined into the task object:

```
{
  "jsonrpc": "2.0",
  "id": 8,
  "method": "tasks/get",
  "params": {
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840"
  }
}
```

```
{
  "jsonrpc": "2.0",
  "id": 8,
  "result": {
    "resultType": "complete",
    "taskId": "786512e2-9e0d-44bd-8f29-789f320fe840",
    "status": "completed",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:50:00Z",
    "ttlMs": 3600000,
    "pollIntervalMs": 5000,
    "result": {
      "content": [
        {
          "type": "text",
          "text": "Hello, Luca!"
        }
      ],
      "isError": false
    }
  }
}
```


Error Handling

Tasks use two error reporting mechanisms:

1. **Protocol Errors**: Standard JSON-RPC errors for protocol-level issues
2. **Task Execution Errors**: Errors in the underlying request execution, reported through task status

Protocol Errors

Servers **MUST** return standard JSON-RPC errors for the following protocol error cases:

- Invalid or nonexistent `taskId`: `-32602` (Invalid params)
  - Servers **MUST** return this error for `tasks/get`.
  - Servers **SHOULD** return this error for `tasks/update` and `tasks/cancel`.
- Internal errors: `-32603` (Internal error)
- Missing required client capabilities: `-32021` (Missing Required Client Capability)
  - Servers **MUST** return this error for non-declaring clients requesting task notifications on `subscriptions/listen`.
  - Servers **MUST** return this error for non-declaring clients issuing `tasks/get`, `tasks/update`, and `tasks/cancel` requests.

Servers **SHOULD** provide informative error messages to describe the cause of errors.

Example: Task not found

```
{
  "jsonrpc": "2.0",
  "id": 70,
  "error": {
    "code": -32602,
    "message": "Failed to retrieve task: Task not found"
  }
}
```

Example: Task expired

```
{
  "jsonrpc": "2.0",
  "id": 71,
  "error": {
    "code": -32602,
    "message": "Failed to retrieve task: Task has expired"
  }
}
```

Servers are not required to retain tasks indefinitely. It is compliant behavior for a server to return an error stating the task cannot be found if it has purged an expired task.

Task Execution Errors

When the underlying request encounters a JSON-RPC protocol error during execution, the task moves to the `failed` status. The `tasks/get` response **SHOULD** include a `statusMessage` field with diagnostic information about the failure, and **MUST** include the `error` field with the JSON-RPC error.

The `failed` status **MUST NOT** be used to represent non-JSON-RPC errors, such as a tool result that completed with `isError: true`. Errors within the context of a protocol method result **MUST** use the `completed` status with the error details in the `result` field. This maintains a strong separation between protocol-level faults (which use the `failed` status) and other faults.

Example: Task with JSON-RPC execution error

```
{
  "jsonrpc": "2.0",
  "id": 4,
  "result": {
    "resultType": "task",
    "taskId": "786512e2-9e0d-44bd-8f29-789f820fe840",
    "status": "failed",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:40:00Z",
    "ttlMs": 3600000,
    "statusMessage": "Tool execution failed: API rate limit exceeded",
    "error": {
      "code": -32603,
      "message": "API rate limit exceeded"
    }
  }
}
```

Example: Tool call completed with tool error (isError: true)

For tool calls that complete successfully at the protocol level but return an tool-level error (indicated by `isError: true` in the tool result), the task reaches `completed` status with the tool result in the `result` field:

```
{
  "jsonrpc": "2.0",
  "id": 5,
  "result": {
    "resultType": "task",
    "taskId": "786512e2-9e0d-44bd-8f29-789f820fe840",
    "status": "completed",
    "createdAt": "2025-11-25T10:30:00Z",
    "lastUpdatedAt": "2025-11-25T10:40:00Z",
    "ttlMs": 3600000,
    "result": {
      "content": [
        {
          "type": "text",
          "text": "Failed to process request: invalid input"
        }
      ],
      "isError": true
    }
  }
}
```

The `tasks/get` endpoint returns exactly what the underlying request would have returned:

- If the underlying request resulted in a JSON-RPC error, the task uses `failed` status and the `error` field **MUST** contain that JSON-RPC error.
- If the request completed with a result (even if `isError: true` for tool results), the task uses `completed` status and the `result` field **MUST** contain that result.

Reservations

- The `tasks/` method prefix and `notifications/tasks/` notification prefix are reserved for this extension.
- The result-discriminator value `"task"` for `resultType` is reserved for this extension.
- The label `io.modelcontextprotocol/tasks` is reserved for this extension.

Rationale

Unsolicited Tasks vs. Immediate Results

An alternative proposal would have handled the immediate result case individually, and with slightly different preconditions: *If* tasks are supported, *and* the client supports immediate task results, *then* servers may return a regular result in response to a task-augmented request. That version of immediate results looked like a better option at the time, as it implied no breaking changes on top of the initial tasks specification.

However, as we look to move away from stateful protocol interactions and given the current experimental state of tasks in general, it seems worth proposing a somewhat more radical change that reduces the complexity of the overall specification and makes tasks more "native" to MCP at this time. In particular, the choice to allow unsolicited tasks (in *addition* to immediate results) means promoting tasks to a first-class concept intended for all persistent operations, as opposed to being a parallel and somewhat specialized concept.

This happens to align with the proposed SEP-2322, but the two are not coupled with one another.

Splitting Reads (`tasks/get`) and Writes (`tasks/update`)

Earlier drafts of this redesign let `tasks/get` carry `inputResponses` so a single round trip would both submit responses and observe the resulting state. That conflation has costs: it makes the read path non-idempotent (a retried `tasks/get` could re-submit responses), it forces the read path to share the eventual-consistency model of the write, and it complicates intermediaries that want to cache or deduplicate reads. Splitting the methods leaves `tasks/get` as a pure, idempotent read that any layer can cache or replay safely, and confines write semantics — including their eventual-consistency window — to `tasks/update`.

`tasks/update`'s ack-only response shape follows from the same separation: there is no read data the server needs to return that the client cannot get from a follow-up `tasks/get`, and forcing an embedded `Task` into the response would re-introduce the non-idempotency we are trying to avoid. The cost is one extra round-trip per round of input — paid only when the task actually requires a client request.

Task Creation Consistency

The following new requirement is introduced:

> A server **MUST NOT** return `CreateTaskResult` until the task is durably created — that is, until a `tasks/get` for the returned `taskId` would resolve. In eventually-consistent environments, the server **MUST** wait for consistency before responding. This requirement eliminates the need for clients to speculatively poll for task creation.

Unlike `tasks/update` and `tasks/cancel`, task creation is strongly-consistent. This has to be the case to avoid speculative `tasks/get` requests from requestors that would otherwise not know if a task has silently been dropped or if it simply has not been created yet. Conversely, eventual consistency in `tasks/update` and `tasks/cancel` works because the client behavior is not contingent on the results of those operations (the client can continue to poll either way). While consistent task creation does increase latency costs in distributed

systems that did not already behave this way, explicitly introducing this requirement simplifies client implementations and eliminates a source of undefined behavior.

This also aligns with long-running operation APIs in general, which typically require that once an operation is acknowledged, it must be findable via the polling endpoint.

Ack-only Cancellation

In the `2025-11-25` design of tasks, `tasks/cancel` returned a task describing the task's state immediately after the cancellation attempt. That return shape implies a synchronous read — the server must consult task state to populate it — but cancellation is inherently asynchronous in many applications (a separate worker decides whether and when to honor it), so the returned task object would in many cases simply repeat what the next `tasks/get` would show. Reducing `tasks/cancel` to an ack matches the operation's actual semantics: The request is a signal, not a state query. Clients that want to know the post-cancel status do so via `tasks/get` on the same code path they use for all other state observation.

The eventual-consistency on the ack is the same separation as for `tasks/update`: The server may record the cancellation request and respond before the worker has actually transitioned the task, without allowing the client to interpret the ack as strongly-consistent.

While `tasks/update` and `tasks/cancel` use ack-only response shapes for the reasons above, servers **SHOULD** still return errors for clearly invalid requests — such as an unknown `taskId`. The ack-only design is about avoiding synchronous reads of task state in the success path, not about suppressing errors that the server can detect at request time. Returning errors for invalid inputs gives clients a faster signal that something is wrong, rather than forcing them to discover the problem indirectly through subsequent `tasks/get` polls.

Composition with Multi Round-Trip Requests

The following new requirement is introduced:

> Server implementations that use multi round-trip requests in conjunction with task creation (for example, a tool that requires elicitation over `InputRequiredResult` before creating a task) **SHOULD** resolve all MRTR exchanges *synchronously* before responding with a `CreateTaskResult`.

A `tools/call` that supports both MRTR (SEP-2322) and this extension may use them sequentially by sending one or more `InputRequiredResult` exchanges to gather input synchronously, followed by a `CreateTaskResult` to hand off to asynchronous execution. This composition is a consequence of the `resultType` discriminator — each response is independently typed and the client switches behavior based on the value it receives, *without* maintaining any state between the two modes. Prohibiting this would require imposing an artificial constraint with no protocol-level mechanism to enforce it, since the client is unaware that the server will create a task ahead of time.

The two flows maintain separate state despite sharing field names. The MRTR phase ends when the server returns any non-`"input_required"` `resultType`, at which point its `inputRequests` keys are consumed. The task phase begins with `CreateTaskResult` and maintains *its own* `inputRequests` keys independently. Key uniqueness for task `inputRequests` is scoped to the lifetime of the task and does not extend to keys from the preceding MRTR phase. Clients do not need to deduplicate across the two flows.

Backward Compatibility

The experimental tasks feature in the `2025-11-25` release is **not wire-compatible** with this extension. Implementations that need to interoperate with both surfaces can shim at the SDK level by implementing the

experimental and extension flows in parallel and dispatching on the negotiated protocol version and the client capability the peer declared. The following table summarizes the expected behavior for each permutation:

Protocol Version	`tasks.*` (legacy)	`io.modelcontextprotocol/tasks`
`2025-11-25`	Legacy experimental tasks per the `2025-11-25` specification. The client opts into task augmentation per request via the `task` parameter on `CallToolRequest`; the server uses `tasks/result`, `tasks/get`, `tasks/cancel`, and (where supported) `tasks/list` per that specification. This extension does not apply.	This extension is not defined under the `2025-11-25` protocol version. Servers **MUST NOT** treat this capability as enabling tasks under that protocol version; requests proceed as if the client had declared no task capability at all.
`2026-06-30`	The legacy capability is not part of this extension. Servers **MUST** treat clients declaring only the legacy capability as non-declaring with respect to this extension. Servers that simultaneously support the `2025-11-25` Tasks specification alongside this extension **SHOULD** continue to permit `tasks/get` and `tasks/cancel` requests from such clients to operate on tasks created under that flow.	The canonical case. Full task lifecycle as specified in this document, with the following wire-level differences from the `2025-11-25` experimental feature: • `tasks/result` is removed; clients calling it **MUST** receive `-32601` (Method Not Found). • The `task` parameter on `CallToolRequest` is removed; servers **MUST** ignore it (treat the field as unknown) rather than using it as an opt-in. • The `tasks.requests.*`, `tasks.cancel`, and `tasks.list` capability declarations are not part of this extension. Servers that previously advertised these **MUST** migrate to declaring `io.modelcontextprotocol/tasks`, and **MUST NOT** continue to advertise the legacy capabilities under any protocol version that includes this extension.

A server that returns the standard `CallToolResult` shape — i.e., never elects to create a task — remains fully spec-compliant under this extension. Clients that have negotiated the extension **MUST** handle both result shapes for any augmented request.

Security Implications

- **Task ID unguessability.** A server **MAY** use task IDs as bearer tokens for a server's stored state. Servers **MUST** generate them with sufficient entropy that a third party cannot enumerate or guess them.
- **Auth binding.** Servers **MUST** perform authentication and authorization checks on each task-related request to ensure that the client has permission to access a task.

- **Cross-caller correlation.** Because there is no `tasks/list`, a server cannot inadvertently leak the existence of one caller's tasks to another. This is an improvement over the `2025-11-25` tasks specification, in which a poorly-scoped list could expose unrelated task IDs.
- **Input-request trust model.** `inputRequests` carry elicitation and sampling payloads from the server through the client to the user or model. Hosts **MUST** apply the same trust model to these payloads as they would to standard elicitation/sampling requests. A task is not a higher-trust channel.

Reference Implementation

Implemented in mcpkit (see usage example).