Tab 1

MCP Interfaces RFC

Authors: Toby Padilla (@toby)
Date: **Apr 3, 2025**

This RFC proposes a new primitive within the Model Context Protocol (MCP): **Interfaces**. An interface is a named, versioned collection of one or more tool method specifications that represent a coherent capability or use-case. Each method includes:

- A canonical method name
- A tool description
- A full JSON Schema for parameters and return values

MCP Servers may implement zero or more interfaces. By doing so, they guarantee that the listed methods exist, with argument and return signatures matching exactly, and consistent descriptions that can be relied upon by downstream consumers.

This design unlocks:

- Fine-tuned LLMs with reusable API surfaces
- Strong client-side discoverability and composability
- Lightweight UI rendering without needing to introspect tools
- Reduced burden on server authors to create meaningful tool descriptions

Definitions

- **Interface**: A named, versioned specification of one or more method signatures including arguments, return schemas, and human-readable descriptions.
- **MCP Server**: An implementation that provides tools. It may implement zero or more interfaces.
- **Tool**: A callable method exposed by a server, which may be part of zero or more interfaces.
- **Method**: A single tool definition within an interface, including its full schema and description.

Use-Case Based Interfaces

Interfaces group related functionality under a versioned name, defining a consistent capability.

Example: `media.player.v1`

```
{
```

```
  "interface": "media.player.v1",
  "description": "Standard interface for controlling playback of
audio/video media.",
  "methods": [
    {
      "name": "play",
      "description": "Begin playback of the current media item.",
      "parameters": {},
      "returns": {
        "type": "object",
        "properties": {
          "status": { "type": "string", "enum": ["playing"] }
        },
        "required": ["status"]
      }
    },
    {
      "name": "pause",
      "description": "Pause current playback.",
      "parameters": {},
      "returns": {
        "type": "object",
        "properties": {
          "status": { "type": "string", "enum": ["paused"] }
        },
        "required": ["status"]
      }
    },
    {
      "name": "set_volume",
      "description": "Set the playback volume.",
      "parameters": {
        "type": "object",
        "properties": {
          "volume": {
            "type": "number",
            "minimum": 0.0,
            "maximum": 1.0
          }
        },
        "required": ["volume"]
      },
      "returns": {
```

```
        "type": "object",
        "properties": {
          "success": { "type": "boolean" }
        },
        "required": ["success"]
      }
    }
  ]
}
```

Other examples:

- `code.editor.v1`: `open_file`, `edit_line`, `run_code`
- `filesystem.navigator.v1`: `list_dir`, `open_file`, `delete_file`
- `chat.assistant.v1`: `summarize`, `reply`, `search_docs`

Interfaces are composable and versioned for long-term stability.

Tool Selection Benefits

Fine-Tuning on Interfaces

Interfaces provide a **canonical API surface** for model training and fine-tuning. Rather than training on many bespoke tool descriptions with slight variations, a model can be trained once on the methods in `media.player.v1`, and generalize to any conforming MCP server.

This is only possible because:

- The method names and argument schemas are fixed
- The **method descriptions are standardized** and meaningful
- Tools appear in consistent combinations that define a coherent capability

This avoids overfitting to specific backends and promotes robust generalization.

No Burden on Server Authors

Tool descriptions are often vague, redundant, or inconsistent across servers. With interfaces, descriptions are included in the interface definition — not on each server.

This means:

- MCP servers do not need to invent their own tool names or descriptions
- The ecosystem becomes more interoperable and predictable

- LLM prompting quality increases due to consistent method semantics

Prompt Context Partitioning

The `list_interfaces` method on a MCP server returns all the necessary information for tool invocation — including method names, input schemas, output schemas, and canonical descriptions.

This allows LLMs and clients to **load only the relevant subset of tools**.

As a result:

- The injected tool context is small and semantically unified
- Models perform better due to reduced ambiguity
- No second-round tool metadata resolution is required

Dynamic Tool Use

MCP Clients can dynamically transition between interfaces during a session. For example:

1. Begin with `filesystem.navigator.v1` to locate a file
2. Use `code.editor.v1` to edit it
3. Transition to `media.player.v1` to play output

This allows LLMs to compose **toolchain-style workflows** where each stage adheres to a distinct, reusable interface.

Registry: Interface Compliance

Registries may index servers based on which interfaces they implement. This allows:

- Discovery of servers by capability
- Visibility into interface adoption
- Clear communication of compatibility to clients

Registries may also store canonical interface definitions for reference, validation, and documentation purposes.

Versioning

Interfaces must be versioned explicitly, following the format `<name>.v<major>` (e.g. `media.player.v1`).

- Adding optional fields: OK in same version
- Changing required fields, types, or semantics: requires a new major version
- Deprecating interfaces: must retain historical version for compatibility

MCP servers declare which interface versions they implement:

```
{
  "interfaces": ["media.player.v1", "code.editor.v2"]
}
```

Clients may request the highest mutually supported version.

Protocol Additions

`list_interfaces`

A new method for discovering which interfaces a server implements, along with their full method specs:

```
{
  "method": "list_interfaces",
  "response": [
    {
      "interface": "media.player.v1",
      "methods": [
        {
          "name": "play",
          "description": "Begin playback of the current media item.",
          "parameters": {},
          "returns": {
            "type": "object",
            "properties": {
              "status": { "type": "string", "enum": ["playing"] }
            },
            "required": ["status"]
          }
        }
      ]
    }
  ]
}
```

This enables both human and programmatic inspection of a server's capabilities without additional calls. Tools that aren't part of an interface, can return under a reserved `"interface": "unclassified"` section, to eliminate the need for a second `list_tools` call.

Schema Definition

Each interface follows a structured schema that includes:

- Interface name and version
- Interface description, can be used for vector retrieval
- A list of methods, each with:
  - `name`: canonical method name
  - `description`: natural language instruction
  - `parameters`: JSON Schema describing inputs
  - `returns`: JSON Schema describing outputs

This makes interfaces ideal for **LLM execution planning**, **UI rendering**, and **schema validation**.

Custom UIs

MCP clients can render bespoke interfaces based on the known method sets:

Interface	**UI Elements**
`media.player.v1`	Play, Pause, Volume Slider
`code.editor.v1`	Syntax-highlighted editor, Run button
`filesystem.navigator.v1`	File explorer, Breadcrumbs

Because all methods in an interface have known names, parameters, return types, and descriptions, **no introspection or prompting is required** to render an effective UI.

Interfaces thus decouple LLMs from the burden of layout logic, while still enabling deeply interactive systems.

Security and Validation

- Servers retain full control over tool access and execution.
- Interface validation is encouraged:

    - Servers may test whether their methods match the parameter and return schemas
    - Registries may optionally perform conformance checks

Interfaces are declarations of **intent and structure**, not enforcement mechanisms — but tooling can validate adherence when needed.

Conclusion

Interfaces provide a foundational abstraction layer for the Model Context Protocol. They enable standardization, composability, and efficiency across LLMs, tools, and UIs. By embedding descriptions and full schemas in interface definitions:

- Models can be fine-tuned on reusable, predictable APIs
- Clients avoid redundant schema lookups
- Server authors reduce boilerplate
- UIs can render declarative layouts
- Tool ecosystems gain structure, searchability, and composability

Interfaces are a simple but powerful primitive for the next evolution of LLM-integrated software.

Tab 2