ChatGPT

Implementing MCP Resources for Real-Time CAD Context Updates

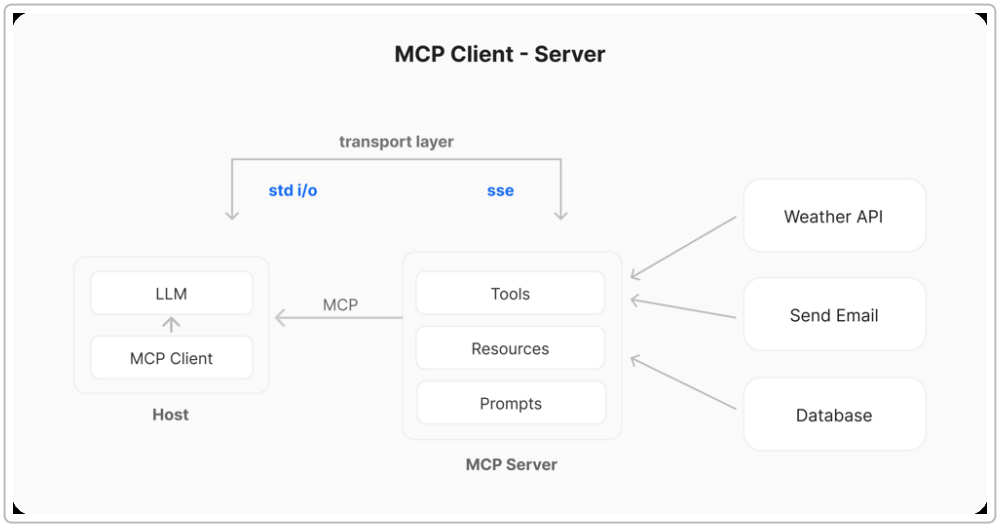

Figure: Simplified MCP client-server architecture. The host (LLM + MCP client) connects to an MCP server that exposes Tools, Resources, and Prompts representing external systems [1].

Overview of MCP Resources for CAD Context

Model Context Protocol (MCP) defines a standardized way to connect AI applications with external tools and data. In MCP, **Resources** are meant for providing structured context data to the AI model [2]. Unlike Tools (which the model calls to perform actions), Resources are **application-controlled** data sources that the client shares with the model (similar to a read-only REST API endpoint) [1]. This makes the Resources component ideal for feeding the AI assistant information about the user's current CAD session – such as the open file's metadata, a log of recent commands, or other contextual details – without executing any command. By leveraging Resources in your MCP server, you can keep the AI up-to-date on what's happening in the user's CAD file in real time. The host application (e.g. Claude Desktop or another AI client) can retrieve these resources as context for the model, and ideally **subscribe to changes** so that updates are reflected promptly for the AI.

How it works: In practice, your Rhino plugin (as the MCP server) would maintain a **"Resources" data structure** containing the current state of the CAD session – for example, a command history log and key metadata like file name, active layers, object counts, etc. The AI client will discover and load these resources when it connects (during MCP handshake and discovery) [3] [4]. After that, whenever the user modifies the CAD model, the plugin can update the resource data (e.g. append the latest command to the history, update the object list, etc.). The MCP client can then fetch the updated resource on demand or, better yet, be notified of changes. MCP supports **stateful connections** and even server-initiated messages (especially when using transports like HTTP with Server-Sent Events) [5]. This means your server can push an event to

the client when a resource changes, allowing the client (Claude Desktop or your platform) to automatically refresh that context. In essence, the AI client "subscribes" to resource updates, enabling a live feed of CAD context.

Monitoring User Operations in Rhino

To keep the Resources in sync with user actions, the **Rhino plugin needs to monitor user operations**. Rhino's API provides event hooks and command tracking that you can leverage. For example, there are events like `Rhino.Commands.Command.BeginCommand` / `EndCommand` and `RhinoDoc.AddRhinoObject` which notify when a command starts or when a new object is added to the document [6] [7]. By handling these events (and others such as object deletion or modification events), your plugin's background routine can detect each meaningful change. On each such event, the plugin would update the relevant resource data structure – e.g. logging the command name and any inputs/outputs to a **command history list**, or updating metadata like object count. The **command history** could simply be a list (capped at a certain length to avoid overflow) that the resource exposes. Rhino's own command log is ephemeral (it isn't saved with the file and even gets cleared during sessions [8]), so maintaining this in the MCP resource as a temporary session log is reasonable and aligns with Rhino's behavior. The resource might expose endpoints or keys like `cad://history` for recent commands and `cad://metadata` for file info, which the client can query. In fact, community MCP servers follow similar patterns (for instance, a shell-history MCP server defines resources like `history://recent` to fetch recent commands [9]).

Designing Resource Updates and Subscriptions

With the plugin updating the Resources backend, the **host AI application can stay in sync** by retrieving these resources at appropriate times. Ideally, the client would get updated context **without user intervention** whenever something changes. There are a couple of ways to achieve this:

- **Polling:** The simplest approach is for the MCP client (Claude Desktop or your app) to re-fetch the resource at key moments (e.g. before each new AI query or at a fixed interval). However, polling can be inefficient or introduce lag.

- **Push via Events:** A more robust approach is to utilize MCP's support for server-initiated messages. When using a persistent connection (such as the `stdio` transport or HTTP + SSE), the MCP server can send notifications to the client proactively [5]. For example, after the plugin updates the command history resource, it could send an SSE event or a JSON-RPC notification indicating "resource X updated". The client, upon receiving this, would then fetch the new resource data and include it in the AI's context. This effectively acts like a **subscription** – the host app is informed of changes in real time. Many modern MCP integrations use SSE to push updates from server to client, which is well-suited for streaming events like these [5].

In implementation, you might maintain a WebSocket or SSE channel from the plugin to the MCP client, or use the built-in MCP client session if it supports receiving notifications. The key is that **Resource data in MCP is meant to be dynamic**, and clients are expected to refresh or receive updates as the server provides them. By structuring your resource as a lightweight GETtable endpoint (for example, `cad://commands` returning the latest N commands in JSON), you adhere to the MCP concept that resources provide data

without side effects [1]. The AI model can then utilize this context (e.g. Claude might incorporate the recent commands list into its prompt to understand what the user has done so far).

Handling Multiple CAD Instances in One Session

One concern you raised is **managing multiple CAD instances** during a single user session. This is important: if a user has two Rhino windows or files open, the plugin may be feeding data from both. How do we differentiate and ensure the right context goes to the right place? There are a few strategies:

- **One Active Connection Per User:** The simplest method (and likely your initial plan) is to **limit the MCP connection to a single Rhino instance per user at any given time**. In practice, this means the user can only pair the AI client with one Rhino session at once. This avoids any ambiguity – all resource updates clearly pertain to that one open file. Indeed, MCP's design typically assumes a 1:1 relationship between an MCP client and server [4]. For example, Claude Desktop or Cursor spawns one client connection for each configured server integration. If you only register one Rhino MCP server in the client config, the user will naturally use it with one Rhino session. This approach is user-friendly and mirrors how most plugins work (e.g. you connect to the currently active document). You could enforce this by having the plugin only allow one session to call `mcpstart` at a time, or by the external app refusing a second connection attempt.

- **Unique Session Identifiers:** If supporting multiple concurrent instances becomes necessary, you'll need a way to tag and separate each instance's data. Using the **file name or file path** as an identifier is one idea. For example, your resource URIs could incorporate the file name: e.g. `cad://<filename>/history` and `cad://<filename>/metadata`. The host client (and user) would then specify or select which file's context to use. However, file name alone can be ambiguous in edge cases – two files with the same name in different directories, or unsaved documents (which might all be "Untitled"). Relying on the full file path can differentiate saved files, but unsaved ones have no path. The Rhino API does provide a **runtime document ID** (a serial number) for each open document, but it's only unique for the running session and not persistent [10]. A more robust solution could be to generate a GUID for each session or document and store it in the document's user data, but that has its own complexities (for example, copied files would duplicate the GUID unless handled [11]).

Given these complications, many developers choose the simpler route of one active instance at a time, at least initially. You mentioned possibly prompting the user to distinguish files with the same name – that could be a friendly prompt if the situation arises (e.g., "You have two documents named 'HouseModel'. Please activate or rename one to use with the AI."). This not only helps the system but also encourages the user to keep their sessions distinct, which can be beneficial for their own clarity.

In summary, **for now it's reasonable to attach the MCP resources to the single active Rhino document** (or whichever document was launched with `mcpstart`), and restrict external AI apps to connect to one session at a time. This avoids confusion and ensures the Resource context truly reflects the user's current focus. As your platform grows, you can revisit multi-instance support, potentially by spawning multiple MCP server instances on different ports (one per Rhino instance) and having the client manage multiple connections. But that's an advanced use-case; many MCP clients (like Claude Desktop) might not even have UI for multiple simultaneous server contexts yet.

Temporary History vs Persistent Metadata

Another point you raised is differentiating **temporary vs. persistent data** in the resources. This distinction is wise:

- **Command History (Temporary):** The stream of commands a user executes is mostly useful *during* an active session to give the AI context. It doesn't need to be saved to disk long-term. In fact, as noted, Rhino itself does *not* permanently log commands – the command window history is ephemeral and even within one session it has a limited buffer [8]. Therefore, treating the command history Resource as a volatile, in-memory list is perfectly acceptable. You might initialize it empty (or with a brief recent history if you can fetch it) when `mcpstart` is run, and clear it when the session ends or the file is closed. This also avoids privacy issues of storing potentially sensitive command sequences. If the user saves and closes the file, it's reasonable for the next session to start fresh (since the AI won't need last session's commands unless specifically required).

- **File Metadata (Persistent):** Information like the file name, path, author, or other metadata could be considered more persistent. You may include some of this in a resource (e.g., `cad://metadata` might list the file name, unit system, creation date, etc.). Some of that persists with the file itself (like units or maybe custom attributes). If you want, you could cache certain metadata on the server side to remember across sessions – but often it's not necessary because the data can be pulled from the file each time. What might be more useful is ensuring that the **identifier for the session** (as discussed above) remains consistent while the session is live. For example, if you generate an ID for the session when a user starts MCP, keep that until they disconnect; don't regenerate it on every minor change. Persistent storage could also come into play if you later allow **session restore** or logging of AI interactions per file, but that's outside the core MCP resources concept.

In practice, implementing the resource as "read-only" from the AI's perspective means you can mix transient and persistent info in it as needed. The AI will just use whatever is currently there. For instance, your `Resources` JSON could look like:

```
{
  "filename": "HouseModel.3dm",
  "filePath": "C:\\Projects\\HouseModel.3dm",
  "objectCount": 42,
  "lastCommand": "_Move",
  "commandHistory": [
     "_Line (points: ...)",
     "_Circle (center:..., radius: ...)",
     "_Move (selected 3 objects)"
  ]
}
```

Here `filename` and `filePath` persist until the file is changed, `objectCount` updates with each add/delete, and the `commandHistory` grows and maybe trims older entries over time. If the user closes the file or ends the session, you drop this data.

Extending Resources to User-Provided Files

Looking ahead, you mentioned allowing **user-provided local files (3D assets, etc.) as resources**. MCP's resource mechanism can indeed handle that, but you must do so carefully. Essentially, you'd be exposing file content to the AI model. This is plausible – for example, one might have a resource like `cad://attachment/{fileName}` which returns the contents of a user's local 3D asset or a summary of it. Some considerations:

- **User Consent:** According to the MCP spec, hosts must get explicit user approval before sharing user files or data via resources [12]. In a desktop integration, this could mean the user explicitly chooses a file to "attach" as a resource for the AI. This ensures privacy and security (the AI shouldn't arbitrarily read files without permission).

- **Data Format and Size:** 3D assets can be large (many MBs) and often binary. Instead of sending the raw model data, you might have the resource provide a *description* or *metadata* of the asset. For instance, if the user drags in a 3D component, the resource could list it in a "available assets" list that the AI can query (with properties like name, dimensions, etc.), and then the AI could ask to place it via a Tool. Alternatively, if the AI truly needs to analyze the file, you might convert it to a textual representation (like a list of objects, or a special summary). Keep in mind current LLMs can't directly parse binary geometry; you'd likely need to extract salient info for them.

- **Resource Lifecycle:** If a user adds a file as a resource, decide if it's only for that session or if it persists. Possibly it's just for the active session (the user might remove or change it later). Implement commands or UI to manage these attached resources (e.g. "Attach model as reference for AI").

Implementing this could be as simple as adding another entry in your Resources data that holds the content or summary of the user's file. The architecture you're planning – where the plugin streams operations and updates to the server's resources – can be extended to also stream file data or notify when a new file resource is added. The AI client would then include that information in the context it provides to the model. This way, the model could, for example, know that "a file named Tree.3dm is available as an asset" and possibly request a tool to insert it into the scene if asked.

Conclusion

Leveraging the MCP Resources component for updating the AI about the user's CAD session is a powerful approach that aligns with MCP's design goal of providing **contextual data** to the model [2]. By having the Rhino plugin monitor user actions and maintain an up-to-date resource (command history, metadata, etc.), you create a live link between the CAD environment and the AI's understanding of it. The host application (Claude Desktop or any MCP client) can retrieve this context and even get live updates through event-driven notifications, ensuring the AI's responses consider the latest user operations.

When implementing this, it's important to handle identification of sessions carefully – initially favoring a one-instance-per-user model to keep things simple, and tagging resource data with a unique ID (like the file name or a GUID) if expanding to multiple instances in parallel. Command history data can be treated as ephemeral session context (since Rhino itself doesn't permanently log commands [8]), whereas key

metadata and any user-attached files can be managed more persistently (with proper user consent for sharing files [12]).

Overall, your plan fits well with how MCP is intended to be used: **Resources** are indeed meant for things like file contents or history logs [13], and using them to reflect a CAD model's state will enable more intelligent and context-aware assistance from the AI. By continuously syncing the CAD state to the MCP Resources, you ensure the AI always has the **latest context** about the user's project, which can greatly improve the relevance and accuracy of its help. Keep an eye on performance (updating and transmitting only what's necessary) and security, but otherwise this approach should provide a solid backbone for AI-assisted CAD workflows.

Sources: The MCP specification and community resources confirm the role of Resources as contextual data providers [2] [1]. Rhino's API supports event-driven tracking of user actions [6] [7], and Rhino's own command log behavior justifies keeping history in-memory [8]. The MCP architecture allows for pushing updates via persistent connections [5], which aligns with the idea of subscription to changes. Consideration of unique identifiers is informed by Rhino's handling of document IDs [14]. These references underscore the feasibility and best practices for the proposed design.

---

[1] [3] [4] [5] Model Context Protocol (MCP) an overview
https://www.philschmid.de/mcp-introduction

[2] [13] Overview - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18/server

[6] Command.BeginCommand event - Rhino developer
https://developer.rhino3d.com/api/rhinocommon/rhino.commands.command/begincommand

[7] RhinoDoc.AddRhinoObject event - Rhino developer
https://developer.rhino3d.com/api/rhinocommon/rhino.rhinodoc/addrhinoobject

[8] Do Rhino files save the log of the commands used? : r/rhino
https://www.reddit.com/r/rhino/comments/18zhz96/do_rhino_files_save_the_log_of_the_commands_used/

[9] GitHub - rajpdus/mcp-histfile: An MCP(Model Context Protocol) Server for retrieving and sharing your bash/zsh history with MCP Client (Cursor, Claude etc.)
https://github.com/rajpdus/mcp-histfile

[10] [11] [14] Unique and Persistent Identifier for a Rhino Document - Scripting - McNeel Forum
https://discourse.mcneel.com/t/unique-and-persistent-identifier-for-a-rhino-document/192223

[12] Specification - Model Context Protocol
https://modelcontextprotocol.io/specification/2025-06-18