Talk to Your Model: Connecting Claude & ChatGPT to APS via MCP

We are moving past the era of "Chat with PDF." The next frontier is **Chat with Tools**.

If you have ever tried to paste a 500MB JSON metadata extract from a Revit model into ChatGPT, you know the pain: *"Message too long."* RAG (Retrieval-Augmented Generation) helps, but it is static. It can't query the *current* state of a model or trigger an action.

Enter the **Model Context Protocol (MCP)**.

RAPS now implements an MCP Server, allowing LLMs (like Claude Desktop) to call RAPS commands directly as tools. This turns your CLI into a live agent interface for Autodesk Platform Services.

How It Works

Instead of dumping data into the context window, the LLM has access to a toolkit. When you ask a question, the flow looks like this:

1. **You:** "Find all 'Basic Wall' instances in the 'Hospital.rvt' file that are over 100m²."
2. **LLM:** *Thinking... I need to inspect the model metadata.*
3. **LLM (Action):** Calls raps model:query --file Hospital.rvt --category Walls
4. **RAPS:** Returns a filtered JSON snippet.
5. **LLM (Response):** "I found 12 walls matching your criteria. Here are their IDs..."

Setting It Up

Add the RAPS executable to your Claude Desktop configuration:

JSON

```
{
  "mcpServers": {
    "raps": {
      "command": "raps",
      "args": ["mcp", "start"]
    }
  }
```

```
}
```

Once connected, you don't need to write SQL or complex filter queries. You just ask. This bridges the gap between the rigid, complex data structures of BIM and the natural language interface of AI.

Zero-Click Releases: Building a Revit Plugin Pipeline with GitHub Actions

In our previous article, *"The Manual Tax"*, we discussed the cost of manual deployments. Today, we kill the manual process entirely.

Deploying a Revit Add-in to Design Automation usually involves:

1. Compiling the DLLs locally.
2. Zipping them up (correctly!).
3. Getting a new token.
4. Uploading the AppBundle.
5. Creating a new Activity alias.

If you do this manually, you *will* eventually upload a debug build to production. Here is how to automate it using RAPS and GitHub Actions.

The Workflow

We will create a pipeline that runs on every git push to the main branch.

1. Define the RAPS Config

Ensure your repository has a raps.toml defining your AppBundle structure. This ensures the CLI knows which DLLs to bundle.

2. The GitHub Action

Create .github/workflows/deploy.yml:

YAML

```yaml
name: Deploy to APS
on:
  push:
    branches: [ "main" ]

jobs:
  deploy:
    runs-on: ubuntu-latest
```

```
  steps:
    - uses: actions/checkout@v3

    - name: Install RAPS
      run: curl -fsSL https://rapscli.xyz/install.sh | sh

    - name: Bundle & Update
      env:
        APS_CLIENT_ID: ${{ secrets.APS_ID }}
        APS_CLIENT_SECRET: ${{ secrets.APS_SECRET }}
      run: |
       # Authenticate
       raps auth login --client-id $APS_CLIENT_ID --client-secret $APS_CLIENT_SECRET

       # Build the bundle (handles zipping automatically)
       raps da:bundle create --path ./my-plugin

       # Update the AppBundle in the cloud and alias it to 'prod'
       raps da:appbundle update my-plugin --alias prod
```

Why This Matters

With this script, your "time to deploy" goes from 20 minutes of clicking to **0 minutes**. You push code, you go get coffee, and your Design Automation pipeline is updated.

Why We Rewrote the Toolchain: Node.js vs. Rust for 5GB Files

"Why not just use the official Node.js SDK?"

We get asked this a lot. The official SDKs are great for web servers. They are terrible for heavy CLI data processing.

In the AEC industry, "Big Data" isn't a buzzword; it's a daily reality. A single Revit file translation can result in a metadata JSON file exceeding 2GB.

The Memory Problem

We ran a benchmark parsing a **3.4GB JSON metadata extract** from a large stadium model.

- **Node.js Script:** Crashed with FATAL ERROR: Ineffective mark-compacts near heap limit. Even with --max-old-space-size=8192, the Garbage Collector (GC) pauses made the process take **4 minutes**.
- **RAPS (Rust):** Processed the stream in **14 seconds** using constant memory (<100MB).

Safety at Compile Time

The other advantage is strict typing. In JavaScript, sending a null instead of an undefined to an APS endpoint might fail silently or return a cryptic 400 error.

Rust's type system forces us to handle every optional field defined in the OpenAPI spec. If the RAPS CLI compiles, the payload structure is correct. We catch the bugs so you don't have to debug them in production.