MCP Dev Summit Bengaluru

Why We Built a CLI Instead of an MCP Server for Jupyter Notebooks

– And What We Learned

MCP Dev Summit Bengaluru 2026

Piyush Jain — Jupyter Maintainer, AI/ML Open Source Team, AWS

The Problem

Jupyter Notebooks Are Everywhere — But Hostile to AI Agents

```
{
  "cells": [{
    "cell_type": "code",
    "source": ["import pandas as pd\n", "df = pd.read_csv('data.csv')\n"],
    "outputs": [{
      "output_type": "execute_result",
      "data": { "text/html": "<table>..." },
      "metadata": {}
    }]
  }]
}
```

- Deeply nested JSON — 30–40% of tokens are structural characters
- Source code stored as arrays of strings
- Outputs carry base64-encoded blobs
- A single misplaced bracket breaks the entire file

The Obvious Path: An MCP Server

What an MCP Server for Notebooks Would Look Like

```
Tools:
  - read_notebook(path) → JSON
  - add_cell(path, source, type) → cell_id
  - update_cell(path, cell_id, source) → success
  - delete_cell(path, cell_id) → success
  - execute_cell(path, cell_id) → output
  - search_cells(path, pattern) → matches
  - connect_server(url, token) → session
  - clear_outputs(path) → success
  ...8-12 tools minimum
```

The problems we foresaw:

- Each tool call = round trip (latency × operations)
- Agent must hold notebook state across calls
- Tool schemas don't teach *workflow* — just API surface
- Multi-cell operations require orchestration logic in the agent

What We Built Instead

`nb` — A Rust CLI with a Skills File

```
# One command, three operations
nb cell add analysis.ipynb --source "$(cat <<'EOF'
@@markdown
# Summary

@@code
print(f"Rows: {len(df)}, Columns: {len(df.columns)}")
df.describe()
EOF
)" && nb execute analysis.ipynb -i -2 -i -1 && nb read analysis.ipynb -i -1
```

The architecture:

1. A fast Rust CLI (`nb`) that follows Unix conventions
2. An AI-optimized output format with line-oriented sentinels
3. An ~800-token skills file that teaches agents *how* to use it

Design Decision #1: The Output Format

Why Sentinels Beat JSON Tool Schemas

````
@@notebook {"format":"ai-notebook","metadata":{"kernelspec":{"name":"python3"}}}

@@cell {"index":0,"id":"f68t57","cell_type":"code","execution_count":1}
```python
df.head()
```
@@output {"output_type":"execute_result"}
```text
   col_a  col_b
0      1      a
```
````

Property	JSON (.ipynb)	Plain Markdown	Sentinel Format
Unambiguous cell boundaries	✓	✗	✓
Token efficient	✗	✓	✓
Graceful truncation	✗	✓	✓
Structural metadata	✓	✗	✓
Activates code training	✗	✓	✓

Design Decision #2: The Skills File

800 Tokens That Replace a Multi-Tool MCP Server

```
# Notebook CLI

Use `nb` for every `.ipynb` operation.

## Core Rules
- Inspect before editing: run `nb read <notebook> --no-output`
- Prefer cell IDs for durable edits. Use indexes for quick positional work.
- Use stdin (`--source -`) for multi-line content to avoid shell escaping.
- Add cells in batches of 3-5 grouped by logical section.
- Every markdown cell must contain a heading AND prose.

## Common Workflows
### Create → Inspect → Edit → Execute → Connect → Manage Outputs
```

What a skills file teaches that tool schemas cannot:

- *When* to use each operation (inspect before editing)
- *How* to compose operations (batch cells by section)
- *Quality constraints* (markdown cells need prose, not just headings)
- *Trust boundaries* (verify the binary, don't paste tokens in logs)

Design Decision #3: Unix Composability

One Shell Command > Multiple Tool Calls

Debugging a failed notebook:

```
# MCP approach: 3 tool calls
result1 = read_notebook("analysis.ipynb")  # full notebook in context
errors = search_cells("analysis.ipynb", errors=True)  # another round trip
output = read_cell("analysis.ipynb", cell_id=errors[0].id)  # another

# CLI approach: 1 shell command
nb search analysis.ipynb --with-errors
```

Composing with other tools:

```
# Find notebooks with import errors across a project
find . -name "*.ipynb" -exec nb search {} --with-errors \; | grep "ImportError"

# Get the Python prefix matching the notebook's environment
$(nb status --python) python -c "import pandas; print(pandas.__version__)"
```

Design Decision #4: Output Externalization

SHA256 Content-Based Hashing

```
@@output {"output_type":"display_data","path":"/tmp/nb-outputs/a3f2b8c9...d4e5.png"}
```

The problem: Notebook outputs include images, HTML tables, LaTeX — content that's either too large or not useful as tokens in an LLM's context window.

The solution:

- Outputs > 4KB automatically externalized to files
- SHA256 content hash as filename (not guessable, deduplicated)
- Absolute paths in sentinel metadata
- 40+ MIME types with JupyterLab-compatible priority ordering
- Agent reads the path from metadata only when it needs the content

Why this matters for agents:

- A 500KB matplotlib plot doesn't consume context window
- Same content always maps to same file (cache-friendly)
- Agent can selectively inspect outputs rather than receiving everything

Real-Time Collaboration ⚠️ Experimental

The One Place We Still Need a Persistent Connection

```
# Connect to a running JupyterLab
nb connect

# Changes appear instantly in JupyterLab
nb cell add experiment.ipynb --source "df.head()"

# Execute via jupyter server
nb execute experiment.ipynb --cell-index -1

# Disconnect when done
nb disconnect
```

How it works:

- Auto-detects running Jupyter servers
- Uses Y.js (same CRDT protocol as JupyterLab) for conflict-free sync
- Connection state persisted in `.jupyter/cli.json`

> **Status:** This is still early. Known bugs around reconnection and kernel state sync — contributions welcome.

Where MCP Would Have Been Better

Honest Assessment: When to Choose MCP

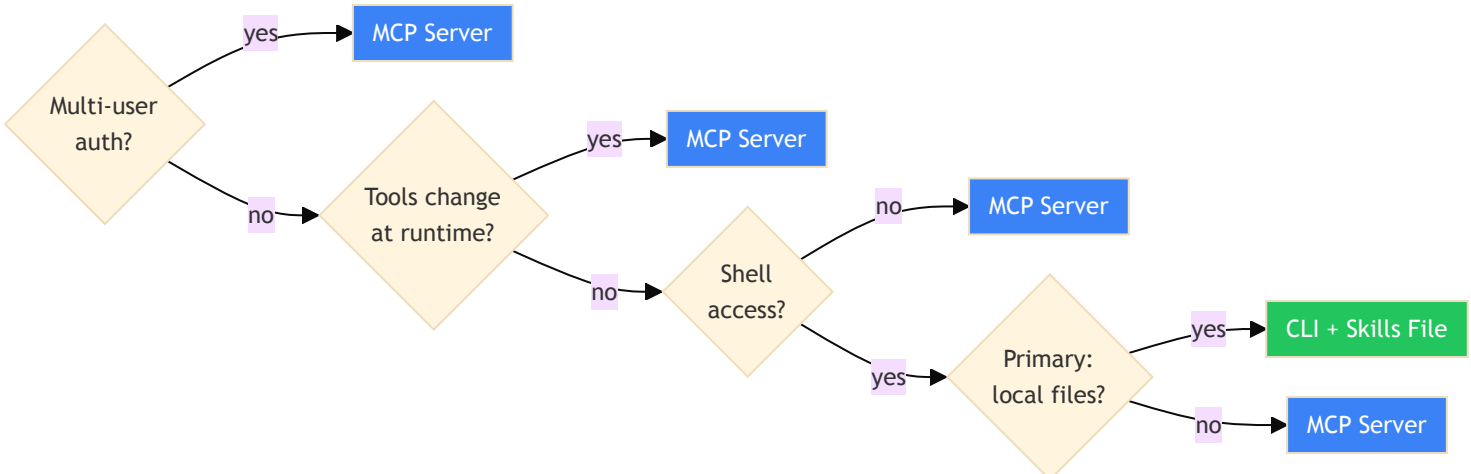
Scenario	CLI + Skills File	MCP Server
Single-user, local notebooks	✓	
CI/CD pipelines	✓	
Multi-tenant auth (JupyterHub)		✓
Dynamic tool discovery		✓
Stateful session management		✓
Cross-language agents (no shell)		✓
Streaming large outputs		✓
Fine-grained permission scoping		✓

MCP wins when:

- You need OAuth/token management across multiple users
- Tools change dynamically based on server state
- The agent runtime doesn't have shell access
- You need streaming responses for large outputs

The Decision Framework

When to Build an MCP Server vs. Ship a CLI + Skills File



Results and Benchmarks

Token Efficiency: CLI vs. Hypothetical MCP Server

Metric	MCP Server (projected)	nb CLI
Tokens to read a 20-cell notebook	~3,200 (JSON response)	~1,800 (sentinel format)
Round trips to add 5 cells	5	1
Context consumed by tool schemas	~2,000 (12 tools)	~800 (skills file)
Agent needs to track state across calls	Yes	No

Real-World Agent Performance

- **Claude** created a complete RL-for-LLMs tutorial notebook (15+ cells) in a single session
- **Codex** fixed 4 bugs in a broken notebook: identified errors, applied fixes, verified execution
- Skills file + references: ~800 tokens core, ~15K tokens total (loaded on demand)
- Works with Claude, GPT, Codex, Gemini — any agent with shell access

Key Takeaways

What We Learned

1. **Format matters more than protocol** — How data appears in the context window determines agent success more than how it gets there
2. **Skills files teach workflow; schemas teach API** — Agents need to know *when* and *how*, not just *what*
3. **Unix composability is an unfair advantage** — LLMs are trained on shell usage; leverage that
4. **MCP and CLIs are complementary** — Use MCP for multi-tenant, stateful, dynamic scenarios; use CLIs for local, composable, file-oriented work
5. **Start with the context window** — Design your agent interface by asking "what will the LLM see?" not "what's the cleanest API?"

```
# Install
curl -fsSL https://raw.githubusercontent.com/jupyter-ai-contrib/nb-cli/main/install.sh | bash

# Install the skill for your agent
npx skills install jupyter-ai-contrib/nb-cli

# Start using it
nb create demo.ipynb
nb cell add demo.ipynb --source "print('Hello, MCP Dev Summit!')"
nb execute demo.ipynb
nb read demo.ipynb
```

GitHub: github.com/jupyter-ai-contrib/nb-cli

Blog: blog.jupyter.org/nb-cli

Piyush Jain — @3coins

Andrii Ieroshenko, Brian Granger — co-developers

MCP Dev Summit Bengaluru 2026