PYHEP.DEV 2026 · 2026-09-07

Model Context Protocol for Scientific Software

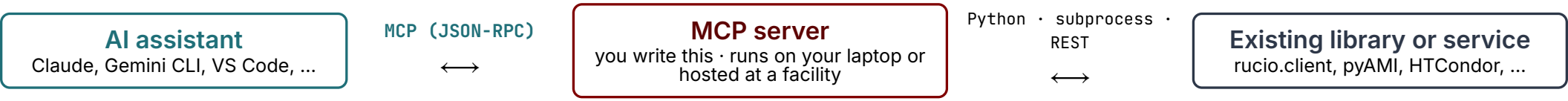
Building AI interfaces that actually work

Giordon Stark (University of Chicago) · everything shown runs in production on the UChicago ATLAS Analysis Facility

Slides AI-assisted by Claude (Anthropic) · title illustration AI-generated. · PyHEP.dev 2026

What is the Model Context Protocol?

An open protocol connecting AI assistants (LLMs like Claude or Gemini) to the libraries, databases, and services scientists already use.



tools functions the LLM calls **resources** data it reads **prompts** reusable templates

modelcontextprotocol.io · not prompt engineering, not a custom foundation model. ↓ a minimal server

Around an existing Python project · backup


```python
from fastmcp import FastMCP
from rucio.client import Client

mcp = FastMCP("rucio")
client = Client()

@mcp.tool()
def list_dids(scope: str,
              pattern: str = "*") → str:
    """Search datasets in a scope by wildcard.

    Use a narrow pattern; results are paginated.
    """
    return ", ".join(
        client.list_dids(scope, {"name": pattern}))

mcp.run()
```

rucio-mcp started from this barebones structure

Everything the LLM sees is right here: **name** `list_dids` , **schema** `scope: str, pattern: str = "*"` , **description** = the docstring

Wrap `rucio.client` , not the CLI: structured data in, curated text out

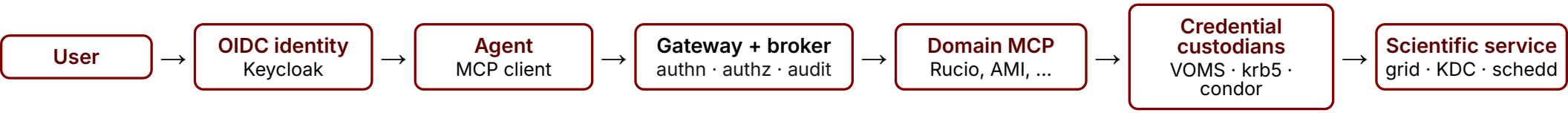
Start on stdio on your laptop; add HTTP when a facility hosts it

`fastmcp` handles the JSON-RPC; you only write the function

The simple picture, and what I actually run

every MCP tutorial

the UChicago AF, in production



THE POINT

The limiting factor is rarely the LLM. It's reliable, secure access to what scientists already use.

One facility, one summer · all in production

7
services behind one gateway

124
tools

6
credential types

~900
users on the UChicago ATLAS AF

The MCP layer

- `rucio-mcp` · datasets, 45 tools
- `ami-mcp` · metadata, 11 tools + a query DSL
- `af-jupyterlab-mcp` · notebooks, 22 tools
- `af-filesystem-mcp` · 4 read-only tools
- `condor-mcp` · batch jobs, 25 tools
- `atlas-search-mcp-bridge` · bridge to the ATLAS OpenSearch MCP

The credential layer

- `af-mcp-platform` · gateway + broker
- `voms-token-service` · grid proxies
- `krb5-token-service` · Kerberos
- `condor-token-service` · HTCondor IDTokens
- `af-credentials` · the shared verify/redeem library

Live gateway snapshot, 2026-09-05 · mcp-portal.af.uchicago.edu.

SIX SERVERS LATER

Lessons learned

interfaces · errors · context · identity · credentials · the facility

An existing API is not an agent interface

bad: the raw API dump

```
{"logicalDatasetName": "mc23_13p6TeV.601229...",
 "crossSection": "0.729", "crossSection_unit": "nb",
 "genFiltEff": "1.0", "kFactor": "1.0",
 "principalPhysicsGroup": "...", "prodsysStatus":
 "...", ...95 more fields, duplicate rows...}
```

every field, no units help, and the same for each of 50 datasets

good: markdown-friendly, LLM-friendly

```
**mc23_13p6TeV.601229** (PhPy8EG ttbar) · VALID
Cross-section: 729 pb (0.729 nb × 1000)
Events: 2M · k-factor: 1.0

---
**Next steps:**
- ami_get_dataset_prov for provenance
- rucio_list_dataset_replicas to locate copies
```

~15 curated fields, units converted in the open, follow-up tools named

> The docstrings keep getting fixed against each tool's actual behavior, to teach the LLM how to use it correctly. The LLM reproduces examples verbatim, so a doc bug is a behavior bug.

Sketch abridged from real ami-mcp output; the curation commit is `901f339` "optimize MCP tools for LLM usefulness".

Errors are the agent's control flow

bad: what the agent sees by default

```
Traceback (most recent call last):
  File ".../replicaclient.py", line 158, ...
rucio.common.exception.DataIdentifierNotFound:
  Data identifier 'user.gstark:mydata'
  not found
```

a dead end: nothing tells the agent what to try next

good: what rucio-mcp returns instead

```
Error: Data identifier
'user.gstark:mydata' not found

**Recovery:** The DID does not exist in
Rucio. Use `rucio_list_dids` with a
wildcard pattern to search for it.
```

returned to the agent, not raised into server logs; the next tool is named

Empty results carry the same routing hints. The error categories also feed my monitoring, so I can see which tools confuse the agent.

> MCP results have an `isError` flag so clients can tell failure from success. I didn't set it at first; do both.

8 The classifier behind the right panel: rucio-mcp `tools/_helpers.py` (`classify_error`).

Context is a bounded resource

bad: `rucio_list_files("mc23...")`, unbounded

```
mc23_13p6TeV:file_000001.root
mc23_13p6TeV:file_000002.root
mc23_13p6TeV:file_000003.root
... 499,997 more lines ...
```

one call floods the LLM's entire context window, and the server downloaded all 500k names to do it

good: capped, and the output says so

```
mc23_13p6TeV:file_000001.root
... 49 more ...

---
Showing 50 results (offset=0).
Pass `offset=50` to see more.
```

a hard server-side limit, announced in the output, with the way to continue spelled out

> And cap the **server's** work too: rucio-mcp once fetched that whole 500k-file container upstream just to serve one 50-row page. Paginate the upstream iterator, not just the output.

The real footer text and iterator fix: rucio-mcp `tools/_helpers.py` (`paginate_iter`).

Identity is ambient, never an argument


```python
@mcp.tool()
async def create_jupyter_server(
    image: str,
    name: str | None = None,
    cpu_cores: int = 2,
    memory_gb: int = 8,
    ...
) → str:
    """...The server is always owned by the caller
    (from their verified broker identity) --
    there is no owner argument. ..."""
    claims = await get_broker_claims(ctx, verifier)
    ... create_notebook(owner=claims.unixname, ...)
```

No tool takes a username. The schema cannot express "read bob's files"

One rule, different ways to implement, e.g.: k8s owner labels · kernel uid/gid · the VOMS nickname in your proxy

Who you are comes from the identity provider (e.g. Keycloak) on every call, not from what a token claims

THE PRINCIPLE

Do not expose dangerous parameters; derive them from verified identity.

af-jupyterlab-mcp `tools/jupyterlab.py` , abridged.

Keep credentials out of the MCP

Lifetime == one call

e.g. a VOMS proxy or Kerberos ticket: fetch it → 0600 temp file → one backend call → delete it. The MCP server keeps nothing.

Exchange, never forward

The user's token never goes downstream (the spec forbids it). Each hop gets a fresh short-lived JWT, valid for **one service**, expiring in **10 minutes**.

One service per secret

e.g. only `condor-token-service` touches the Condor pool key; the broker is its only allowed caller, and it returns short-lived IDTokens.

bad: a token service deciding permissions

```
if "atlas-users" in token["groups"]:
    mint_ticket(user)
# a second copy of the access rules,
# trusting whatever the token claims
```

good: it only verifies who is calling

```
claims = verify(token, aud="krb5-token-service")
mint_ticket(claims["sub"])
# the gateway already decided permissions;
# this token proves who, from the broker
```

sketch; written into all three token services' docstrings (krb5 `app.py`)

> Never store secrets in the MCP server. Keep them in a vault (OpenBao, here) behind the broker, and mint short-lived credentials per call.

The custodians · deep-dive

voms-token-service

The only pod that sees home dirs and Globus passphrases. Runs `voms-proxy-init` impersonated; privilege floor found live, five times.

krb5-token-service

Password → `kinit` → ticket; zero secrets at rest, passwords zeroed on every exit path. Rate limiter protects **CERN's** lockout counter, not itself.

condor-token-service

Runs `condor_token_create` next to the pool key, which "must never reach the broker": it can mint any identity.

WHY THREE SERVICES, NOT ONE

Different privilege floors (root+caps / none / one init container). Merging them forces the union of privileges onto everything. Each is a one-question audit: "did this call come from the broker?"

Orchestrate the hardened native binary; parse artifacts (ccache bytes, PEM), never CLI prose.

The facility between your code and the backend

AF gateway
speaks OIDC / OAuth 2.0 bearer tokens

↓

atlas-search-mcp-bridge (~600 lines)
af-credentials writes the ccache (cached in the broker's vault) to a short-lived file per request

↓

`curl --negotiate`
does the SPNEGO/Kerberos exchange; zero lines of my Python

↓

CERN's ATLAS OpenSearch MCP
Kerberos-only; not mine to change

Common gotchas (each one bit me)

- Kerberos and CA trust config must be in the container (krb5.conf, your CA bundle)
- network policy and DNS have to allow the pod → KDC path
- proxies in between can quietly drop headers

THE PRINCIPLE

Unit tests with mocks can't see any of this. Test against the real facility services before calling it done.

atlas-search-mcp-bridge · SPNEGO = negotiated Kerberos auth at the HTTP layer.

MANY MCPS, ONE ENDPOINT

From servers to a platform

what breaks at ten MCPs · one tool call, end to end

What breaks at ten MCPs (that one never sees)

DESIGN DECISIONS BAKED INTO THE AF MCP PLATFORM

Name collisions: two MCPs serve a tool with the same name; fix = a prefix per service (`rucio-atlas_*` , `fs_*`)

Each service gets its own fixed OAuth audience, pinned in config, never derived from a name that can change

Listing needs credentials too: some MCPs require auth even for `tools/list` , so the gateway mints one just to enumerate

New tools are opt-in: when an MCP releases a tool, it stays disabled until an operator maps it at the broker

Authorization is plain RBAC: Keycloak group → permission → tool

A dead backend disappears from `tools/list` instead of every call failing

ADDING THE NTH MCP

Each MCP is added via config (Infrastructure-as-Code). Sometimes it needs a token service, sometimes a bridge — never changes to the MCP itself.

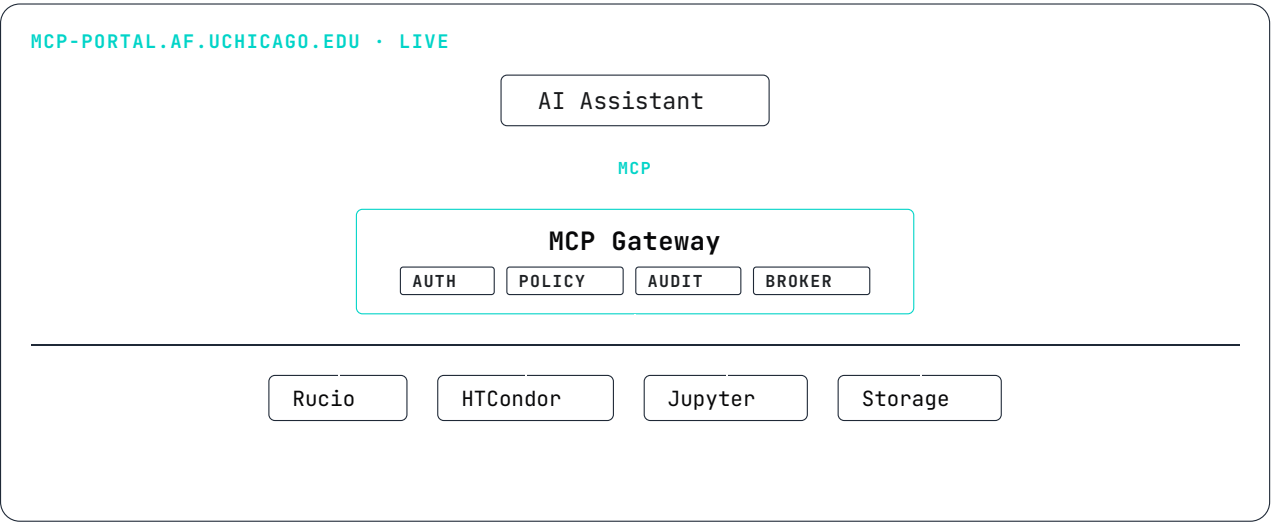
af-mcp-platform issues `#113` , `#257` , `#121` · 289 commits, 77 days, 1,337 tests. ↓ more scars

Tried X, discovered Y, ended with Z · backup

Tried	Discovered	Ended with
Rust CLI wrapper	parsing text tables is fragile	Python + native client
Returning raw API rows	noise swamped the LLM	curated fields, converted units, next steps
Docstrings written from memory	the LLM reproduced broken examples verbatim	docstrings tested against the live service
Stateful MCP sessions, 2 replicas	cross-replica "Session terminated"	stateless HTTP (now the spec default)
Dynamic Client Registration	you maintain a database of client ids, and a restart orphans them all	CIMD (later adopted by the spec)
Groups inside JWT claims	revocation waits for token expiry	live IdP lookup, a real kill switch

Full catalogue with commit hashes: mcp-design-talk.md §7.2 (about 40 of these).

One tool call, end to end



Credentials stay behind the broker

The assistant never holds a raw credential; the user's token is never forwarded.

Authorize every call

Group-based access; denials audited too.

Short-lived, scoped

Minted per service, cached at the broker, reused until expiry; the per-call copy is deleted after use.

mcp-portal.af.uchicago.edu · the portal landing page's own animation (af-mcp-platform). ↓ the same call, step by step

The same call, step by step · deep-dive

1. Claude calls `rucio-atlas_rucio_list_dataset_replicas` with its **personal access token**
2. Gateway authenticates; groups and POSIX identity come **from the IdP (Keycloak)** on every call
3. Authorized once: this tool requires `read_data` ; **denials are audited too**
4. Broker mints a JWT: `aud=rucio-mcp-atlas` (one service), `exp` = 10 min; it says **who you are, not what you may do**
5. rucio-mcp verifies that JWT, then redeems the **VOMS proxy directly from the broker's vault**
6. One Rucio call as the user; proxy deleted; **one audit record**, traced and metered

> The user's token is never forwarded downstream; proxies and tickets never pass through the gateway.

Interfaces that travel

Across LLM providers

Design for the **least-capable LLM client**: e.g. Claude Desktop had no OAuth discovery, so I added personal access tokens; gemini-cli choked on an OAuth metadata fallback, so I deleted it

Standard OAuth, standard transports, no provider-specific hooks

Across experiments

rucio-mcp already serves ATLAS, CMS, DUNE, ESCAPE: the domain layer is VO-agnostic

Transplant the **shape** (broker + custodians + config-only services), not the CERN-specific code

> Interoperable tool interfaces are likely to matter more than LLM-specific integrations.

↓ how my choices scored against the evolving MCP spec

Scored against the MCP spec · backup

My choice	The spec (2026-07-28)	Verdict
Stateless HTTP, state as explicit handles	protocol sessions removed	compliant / anticipated it
CIMD instead of Dynamic Client Registration	DCR deprecated for CIMD	compliant / anticipated it
Tools only; no sampling, roots, logging	all three deprecated	compliant / anticipated it
Exchange tokens, audience-validate everywhere	token passthrough forbidden (MUST)	compliant
Markdown-only outputs, no `outputSchema`	structured content recommended	wrong; retrofitting now
No read-only / destructive tool annotations	annotations recommended	wrong; the gateway must keep its own list of which tools write

Spec: modelcontextprotocol.io/specification/2026-07-28 · full comparison in mcp-design-talk.md §6.3.

When is an MCP server appropriate?

✓ Build one when

- a library or service scientists already use
- assistants need it repeatedly, across clients
- you want to control what context the agent gets injected with
- you can bound outputs and make errors recoverable

× Skip it when

- a one-off script would do
- the backend's API is still churning under you
- the "tool" would hand the LLM a credential

> *Interoperable tool interfaces are likely to matter more than LLM-specific integrations.*

Design the interface for the LLM; design the architecture for the credential. ↓ the checklist

The checklist · backup

One domain per server, with written non-goals: datasets, or metadata, or notebooks; not all three

Docstrings are the LLM's manual: write them for the LLM and test the examples against the live service

Bound every output with a hard server-side limit, and say in the output when you truncated

Errors name the next tool to call, and set MCP's `isError` flag on the result

Structured outputs (`outputSchema`) from day one, not retrofitted like mine

No username parameters: identity comes from the verified token, never from the LLM

No stored credentials in the server: fetch per call from a vault, use once, delete

Permissions decided at one gateway, not re-implemented per server

Stateless server: any state lives in the backend, addressed by an id the tools pass around

Test against the real facility: the failures live in config, trust stores, and network, not in your logic

The long version, with commit-level citations: mcp-design-talk.md.