MCP Dev Summit Mumbai

MCP: The Gateway To Real-Time Human–AI Collaboration in Jupyter at Scale

The Jupyter AI Ecosystem — June 2026

Piyush Jain

Jupyter Server Council Member

Jupyter Distinguished Contributor

Principal Engineer, AI/ML Open Source Team, AWS

Notebooks Are the Go-To Environment for AI

GitHub Octoverse 2025

2.4 million

repositories used Jupyter Notebooks in 2025

+75% YoY

year-over-year growth — the fastest growing file type on GitHub

> "Notebooks are the go-to exploratory environment for AI — fueled by the need to sandbox agents and LLMs."
> — GitHub Octoverse 2025

Jupyter AI v3

The AI integration layer for JupyterLab – rebuilt from scratch for the agent era.

What it is

Jupyter AI is the official sub-project of Jupyter that provides AI integration. v3 is a ground-up rewrite: from a single monolith (LangChain-only, 10s startup) to **10+ independent, composable packages**.

Each package does one thing and plugs into the others via standard Python entry points — no tight coupling, no mandatory dependencies.

Where MCP fits

Jupyter AI v3 exposes **two integration surfaces**:

- **MCP (outbound)** — Jupyter becomes an MCP server. External AI clients call into your live notebooks, kernels, and file system.
- **ACP (inbound)** — External AI agents (Claude, Gemini, Codex…) run as chat personas inside JupyterLab, writing directly to open notebooks.

Both surfaces write to the same live, kernel-connected documents.

You never have to write a notebook from scratch again

Autonomous agents that create, edit, execute, and iterate on notebooks while you focus on the questions that matter.

- AI personas write, debug, and run cells on your behalf
- Full tool access: files, kernels, Git, terminal — all agent-controlled
- Real-time collaboration between you and multiple AI agents

jupyter-server-mcp

A Jupyter Server extension that runs an MCP server — making any Python function callable by AI agents.

How it works

`jupyter-server-mcp` starts a **FastMCP HTTP server** alongside your Jupyter Server. Any Python function becomes an MCP tool — registered two ways:

Via config:

```
# jupyter_server_config.py
c.MCPExtensionApp.mcp_tools = [
    "jupyter_ai_tools.toolkits.notebook:read_notebook",
    "jupyterlab_commands_toolkit.tools:list_all_commands",
]
```

Via entry point (auto-discovered):

```
[project.entry-points."jupyter_server_mcp.tools"]
my_pkg = "my_pkg.tools:TOOLS"
```

Two transports

Stdio — `jupyter-server-mcp` ships a `jupyter-server-mcp-proxy` CLI script that auto-discovers running Jupyter servers and bridges stdio ↔ HTTP. Used for desktop MCP clients.

Direct HTTP — for programmatic access:

```
http://localhost:3001/mcp
```

The Setup

Jupyter becomes the MCP server. External AI clients call tools that run inside your live Jupyter environment.

What You Can Expose as MCP Tools

Any package can register tools via entry points — here are two that ship out of the box.

jupyterlab-commands-toolkit

Bridges MCP calls to 700+ live JupyterLab frontend commands via a bidirectional event system.

`list_all_commands(query?)` Discover all available commands — filter by keyword.

`execute_command(command_id, args?)` Execute any JupyterLab command:

- `notebook:run-all-cells`
- `docmanager:open` + `{"path": "analysis.ipynb"}`
- `filebrowser:toggle-main`
- `notebook:restart-kernel-and-run-all-cells`

> The call goes MCP → Jupyter Server → frontend event → JupyterLab command registry → result back.

jupyter-ai-tools

Four toolkits operating directly on live YDocs — edits are immediately visible to all collaborators.

Toolkit	Operations
fs	read, write, edit, glob, grep
notebook	cell add/update/delete (UUID-addressed)
git	clone, status, log, commit, push
exec	bash execution

> Notebook tools read and write via JupyterLab's CRDT document format — not raw file I/O.

The Foundation: jupyter-server-documents

Server-side document state — the reason MCP tools on Jupyter are more powerful than on generic environments.

What makes it different

In standard JupyterLab, the **browser** holds canonical document state. We flip that — the **server** holds it.

This means:

- Agents can write to notebooks even when **no browser session is open**
- Out-of-band changes (`git pull`, file writes) are detected and reconciled
- Multiple agents and humans edit the **same live document** — never diverging copies

How it works

CRDT-Based — Y.js conflict-free replicated data types merge concurrent edits automatically. WebSocket sync with user cursors and presence awareness.

OutputProcessor — Large cell outputs separated from the YDoc and lazy-loaded, reducing memory pressure on collaborative sessions.

Cell lookup cache — `_cell_indices` with reverse-order scan, optimized for the common agent pattern of appending to the end of a notebook.

MCP Out, ACP In

Two protocols, two directions — both writing to the same live documents.

MCP — outbound

Jupyter becomes a **tool server**.

External AI clients (Claude Desktop, Cursor, your own agent) call into Jupyter to read files, execute cells, run git operations, and control the JupyterLab UI.

Direction: External client → Jupyter

ACP — inbound

External AI agents become **chat personas** inside JupyterLab.

`jupyter-ai-acp-client` runs Claude, Gemini, Codex, and others as subprocesses — wrapped as personas that appear in JupyterLab chat and write directly to open notebooks.

Direction: Jupyter → External agent subprocess

> Both paths write to the same YDoc. An edit from an MCP tool call and an edit from an ACP persona are indistinguishable from the document's perspective.

AI Personas

Customizable AI assistants inside JupyterLab chat — each with their own capabilities, system prompt, and model backend.

The architecture

`BasePersona` — abstract class: name, avatar, async `process_message()` handler, and a `PersonaAwareness` wrapper (pycrdt) for real-time presence.

`PersonaManager` — server-side registry. Discovers persona classes via Python entry points at startup. Stable IDs: `jupyter-ai-personas::pkg::Class`.

`jupyter-ai-router` — dispatches chat messages to the right persona via pycrdt ArrayEvent observers on the YChat document. Slash commands handled separately before reaching any agent.

Built-in ACP personas

`jupyter-ai-acp-client` wraps these as subprocesses — 50-message history context, shared subprocess per agent type, permission/approval workflow for tool calls:

- @Claude — claude-agent-acp
- @Codex — codex-acp
- @Gemini — Gemini CLI
- @Kiro — kiro-cli
- @OpenCode — opencode CLI
- @Goose — goose CLI
- @Copilot — Github Copilot
- @Mistral-Vibe — vibe-acp

+ create your own in < 20 LOC

Build Your Own Agent

Subclass BasePersona — or BaseAcpPersona to wrap any ACP-compatible agent.

1 — Subclass BasePersona

```python
class MyAgent(BasePersona):
    @property
    def defaults(self):
        return PersonaDefaults(
            name="MyAgent",
            system_prompt="You are..."
        )

    async def process_message(self, message):
        # Call any API, run local models,
        # read files, stream back a reply
        self.send_message("Hello!")
```

To wrap an existing ACP agent instead, subclass `BaseAcpPersona` — subprocess management, streaming, permission workflow, and history context all inherited.

2 — Register via entry point

```
[project.entry-points."jupyter_ai.personas"]
my_agent = "my_pkg:MyAgent"
```

3 — Install & chat

```
pip install my-agent-pkg
```

`@MyAgent` appears in JupyterLab chat automatically.

Coming in Persona API v0.1: No-code persona creation via markdown files — drop the file in `.jupyter/personas/`, no install needed.

The Shared Context Bus

MCP tools and ACP personas converge on the same live document layer.

The stack

1. **User / AI Agent** — human in browser or autonomous persona
2. **Jupyter AI Chat + Personas** — @-mentions, slash commands, awareness
3. **MCP + ACP Bridge** — outbound tool server + inbound persona runtime
4. **Tool Registries** — Commands Toolkit, AI Tools, custom entry points
5. **JupyterLab Core** — YDocs, kernels, file system, Git

Why convergence matters

Same document — MCP tool writes and ACP persona writes merge in the same CRDT YDoc. No diverging copies.

Live state — Every tool operates on kernel-connected documents. An MCP `execute_command("notebook:run-all-cells")` triggers real kernel execution and the output lands in the shared document.

Presence-aware — AI agents appear as collaborators with cursors, just like human users.

MCP Dev Summit Mumbai

Demo

Real-Time Human–AI Collaboration in JupyterLab

What's Next

Active work across the ecosystem — better compatibility, simpler persona creation.

jupyter-server-documents v0.3.0

Fixing compatibility with Voilà, nb-cli, and jupyter-collaboration:

- **Outputs service off by default** — v0.2.0 memory improvements make this safe; removes the source of most extension conflicts
- **Remove frontend overrides** — document resets no longer needed; eliminates widget/shared model overrides
- **Remove kernel layer overrides** — upstream fixes where possible; resolves Voilà incompatibility
- **REST API for cell execution** — inspired by Jupyverse; a cleaner, more compatible alternative to kernel client overrides

Persona API v0.1

Making persona creation accessible without writing Python:

- **YAML-fronted Markdown files** — define a persona as a `.md` file with YAML header; drop in `.jupyter/personas/`, no `pip install` required
- **Runtime configuration** — update model, system prompt, MCP servers, and skills without server restart via `update_model()`, `update_context()`, `update_identity()`
- **Per-instance context** — two instances of @Claude can have different skills paths and MCP server configurations
- **New Pydantic models** — `PersonaModel`, `PersonaContext`, `PersonaIdentity` separate identity from capability from configuration

jupyter-ai-contrib Organization

Production-ready building blocks for human-AI collaboration in Jupyter.

- **jupyter-ai-router** — Message routing & slash commands
- **jupyter-ai-persona-manager** — Persona registry & lifecycle
- **jupyter-ai-tools** — File, notebook, Git, bash tools
- **jupyter-ai-chat-commands** — Slash commands to support file attachments and manage personas
- **jupyter-server-documents** — Real-time collab & output mgmt
- **jupyter-server-mcp** — MCP server extension for Jupyter
- **jupyter-ai-acp-client** — ACP persona runtime
- **jupyterlab-commands-toolkit** — Frontend commands as MCP tools

Principles: Production-ready | Composable & modular | Community-governed | BSD-3-Clause licensed