



Open Science and Data Exploration with Jupyter:
Building the Future Together



Max Klein

Jupyter Contributor

Children of Jupyter: What Core Should Steal Back



About me



- I have a background in computational biophysics and simulation. I was a heavy user of IPython Notebook, and now Jupyter.
- While procrastinating on my PhD work I became a major contributor to Jupyter Lab and the larger Jupyter ecosystem.
- These days various entities (currently Capital One) pay me for making and maintaining Jupyter related software.

Do we like notebooks?



- Jupyter's 2011 kernel design has aged well: Colab, VS Code, Deepnote, and even ChatGPT's Python tool all run Jupyter kernels today.
- The notebook itself, though, still runs on the same linear REPL as in 2011, and hidden state remains its most famous complaint.

Do we like notebooks?



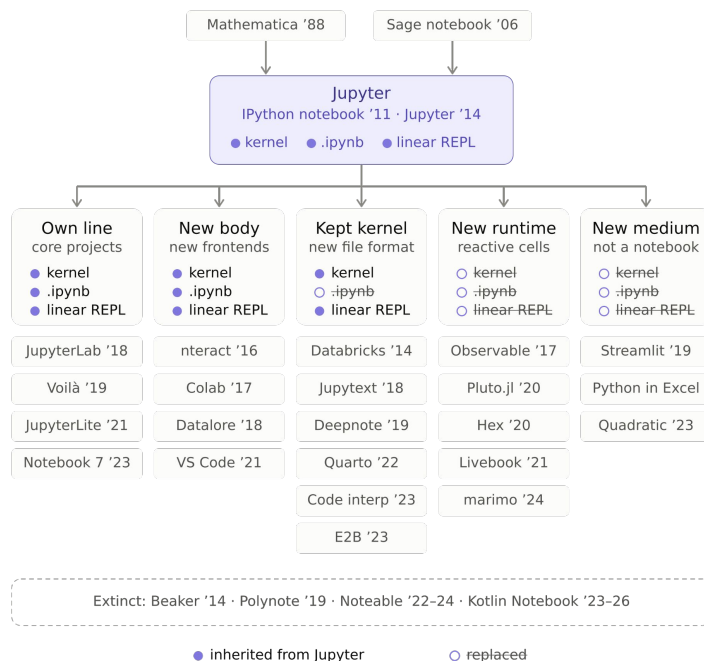
The children of Jupyter



- A generation of descendants (Observable, Pluto, marimo, Hex, Livebook, Deepnote) has spent a decade shipping fixes for these problems.
- It is past time for Jupyter core to steal from its children, but first it needs a survey of what they built.

About this talk

- What this talk is **not**: an attempt to construct a full genealogy of notebook projects.



About this talk



- What this talk is **not**: an attempt to construct a full genealogy of notebook projects.
- Instead this will be a (very brief) survey of the most interesting new features from the “child” projects.
- Ultimately this talk is about which features core should steal back. At the end I'll introduce Jupyter-DAG, an attempt to bring reactivity into core.

Reactive execution

- Reactivity tackles the loudest, longest running complaint about notebooks: hidden state.
- Hidden state goes away if a notebook is “self-consistent” (Macke): its state and its written contents match.
- Reactivity attempts to enforce self-consistency by selectively re-running cells based on a map of cell-to-cell dependencies.

Reactive execution

Reactive execution in Pluto.jl demo

Pluto.jl 

31-pluto/reactive.jl



```
using PlutoUI ✓  
#@bind n Slider(1:20, default = 10, show_value = true)
```



 Settings

 FAQ

 Live docs

 Status

 What do you think about **Pluto.jl**?

Instant feedback...

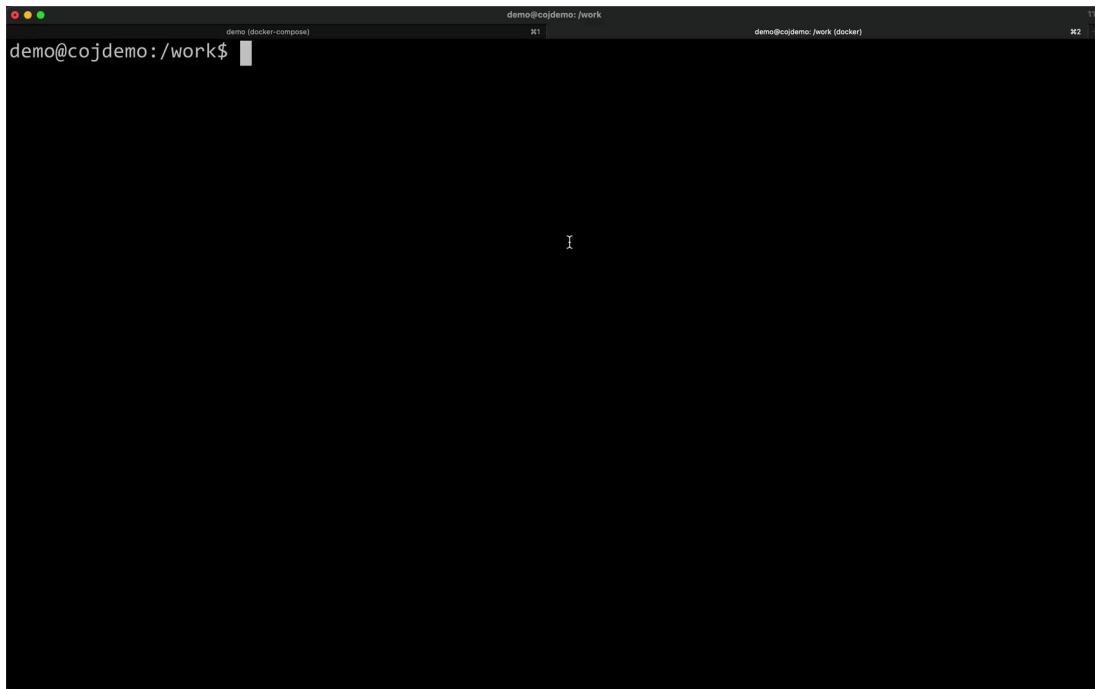
Send

Environment bundled into notebook

- Another common complaint: notebook results are hard to reproduce over time and over different environments.
- Marimo solves this with an optional a pep 723 `# /// script` style header that lists the notebook's dependencies.
- In sandbox mode, Marimo sets up a minimal environment with *only* these dependencies.

Environment bundled into notebook

Marimo sandbox mode demo

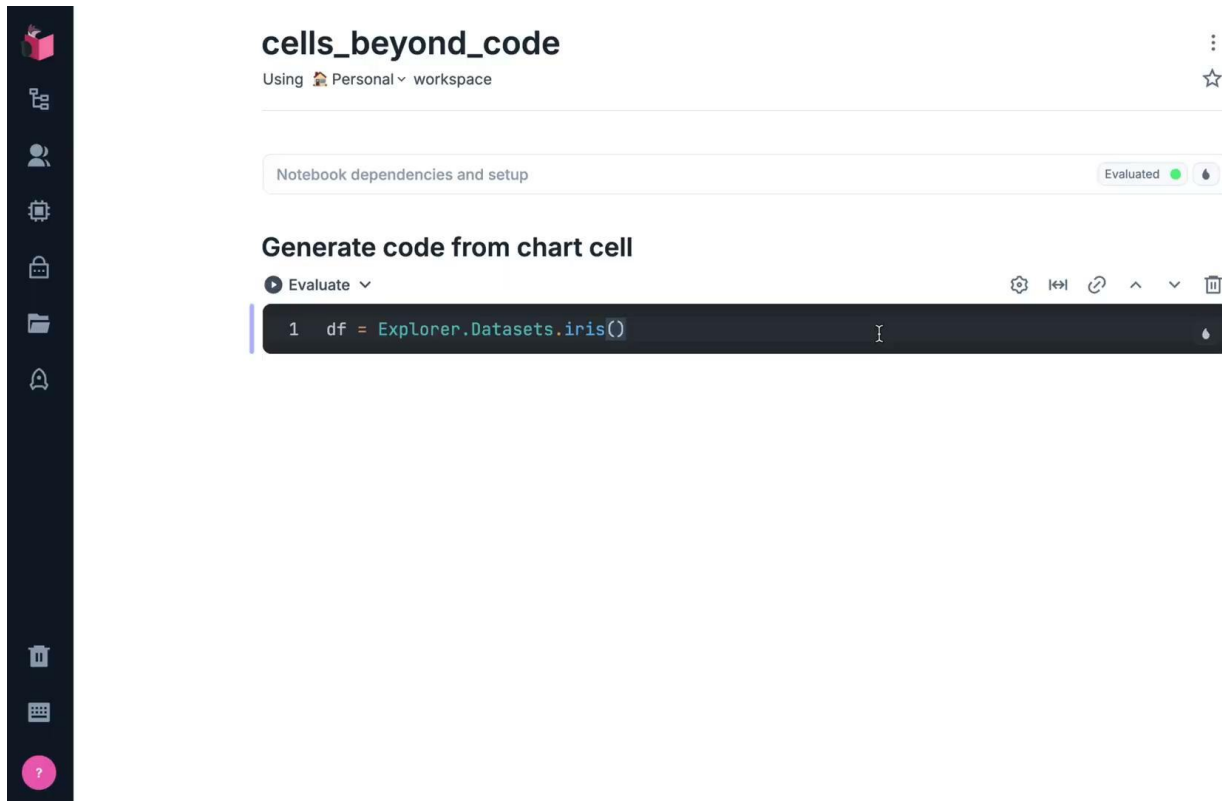
A screenshot of a terminal window titled "demo@cojdemo: /work". The terminal shows a shell prompt "demo@cojdemo: /work\$" with a cursor. The window has a dark grey title bar with standard macOS window controls (red, yellow, green buttons) on the left. The terminal content is black with white text. The prompt "demo@cojdemo: /work\$" is at the top left. A small cursor is visible in the center of the terminal area. The window title bar also shows "demo (docker-compose)" and "demo@cojdemo: /work (docker)" on the right side.

Cells beyond code

- In core Jupyter, cells are code, markdown, or “raw”. But there are many more possibilities.
- Many modern notebooks have SQL cells. Livebook has several other kinds too, including chart cells.
- Chart cells look like chart widgets, but the key difference is that they are input, not output.

Cells beyond code

Livebook chart cells demo



The screenshot displays the Jupyter Livebook interface. On the left is a dark sidebar with icons for file management, search, and help. The main area shows a notebook titled "cells_beyond_code" using the "Personal" workspace. Below the title is a section for "Notebook dependencies and setup" with an "Evaluated" status. The current cell is titled "Generate code from chart cell" and contains a single line of Python code: `1 df = Explorer.Datasets.iris()`. The code cell has a toolbar with options to evaluate, run, or delete the cell. At the bottom right, there are three circular icons: a file icon, a folder icon, and a green circle with a white 'i' labeled "ins".

cells_beyond_code
Using  Personal ▾ workspace

Notebook dependencies and setup Evaluated 

Generate code from chart cell
▶ Evaluate ▾      

```
1 df = Explorer.Datasets.iris()
```

   ins

Novel notebook layouts or Cells... in... space!

Cells in a grid: Marimo multi-column mode



```
1 one = 19
1 two = one**2
1 three = one**1.7
+ 1 four = two*three
+ 2 four
```

PYTHON MARKDOWN SQL GENERATE WITH AI

- You can arrange cells arbitrarily in both vertical and horizontal space without changing the result.

- Execution order comes from per-cell dependencies, so layout doesn't affect computation.

Novel notebook layouts or Cells... in... space!

Cells in a graph: Hex Notebook graph view



- As you type, Hex Notebook builds an equivalent graph showing execution order.
- Hex finds the node dependencies the same way eg Pluto does for its reactivity feature.

Next steps for Jupyter core



- All of these “child” features would be worth bringing into core.
- We should prioritize bringing in reactive execution and environment in notebooks, as these two address some of the most common complaints about Jupyter.
- What’s held reactivity back from core: every implementation so far supports a single language, and Jupyter is language agnostic.

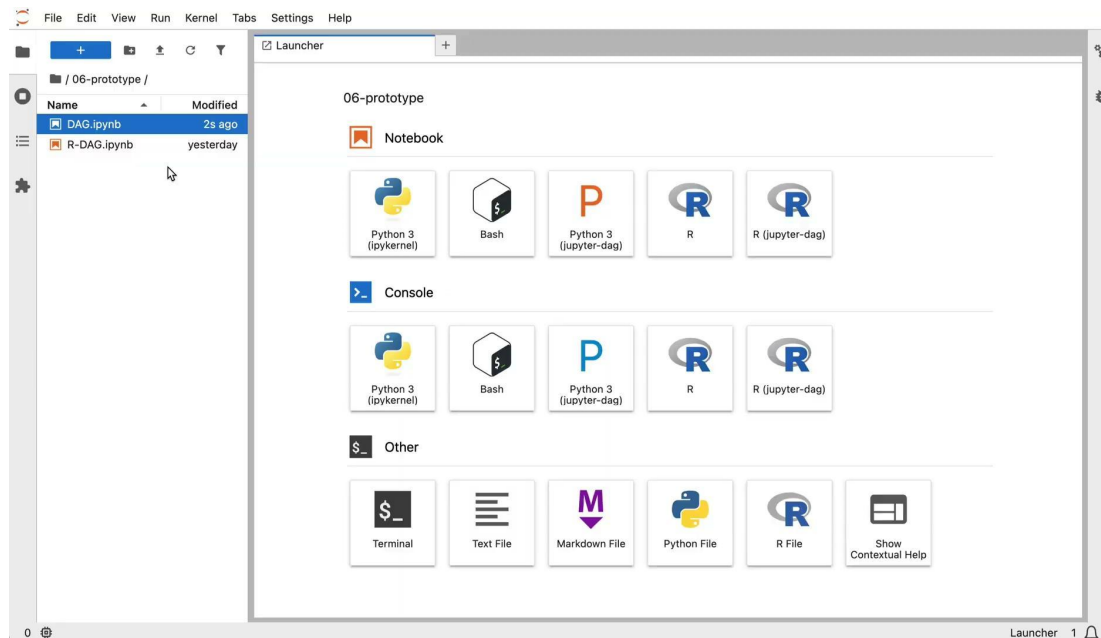
The reactive kernel protocol (RKP)

four additions (all optional, advertised as JEP 92 supported_features)

1. **analyze_request** → **analyze_reply** (shell or control). In: a list of `{cell_id, code}`. Out, per cell: the global names it **defined**, **referenced** and **deleted**, and a **dynamic** flag for effects the parser cannot see, or a parse error.
2. **namespace_delete** on **execute_request**: names to unbind before the code runs.
3. **namespace_set** on **execute_request**: name to JSON value, bound before the code runs.
4. **namespace_delta** on every **execute_reply**: the names the run **added** and **removed**, sent to every connected client, whoever ran the cell.

RKP in practice: the Jupyter-DAG prototype

- Get the latest version here: <https://github.com/telamonian/jupyter-dag.git>



Acknowledgements



- Thanks to Stephen Macke for his work on reactivity and a very helpful conversation at last year's JupyterCon.
- Thanks to my good buddy Claude Fable, which wrote most of the prototype code and helped me edit this presentation.
- Thanks most of all to my new employer Capital One (I just started as a lead developer in August), which has funded this work and my appearance at Jupyter Day.
- Jupyter-DAG: <https://github.com/telamonian/jupyter-dag.git>



Apps, jobs, and agents

“The agent is the user most hurt by hidden state.”

- Jupyter AI (core subproject): started as a chat sidebar, v3 turns it into a host for outside agents. Agents connect over ACP, chats are shared documents, and there's a per-agent permission mode, so core's answer to agents is procedural: ask before you act.
- notebook-intelligence (Mehmet Bektas): a third-party JupyterLab extension with an agent mode that creates, edits and runs cells and tries to fix what breaks, plus MCP servers and local models via Ollama. Shows how much agent tooling lives *outside* core.
- marimo (July 2026, “Notebooks as a tool for agents”): the agent works *inside* the live kernel and commits cell edits as a transaction. Before applying, marimo checks for duplicate definitions, cycles and removed names; a bad batch is rejected whole. The dataflow graph is the agent's type checker.