Python online editors like Theia , polynote, datalore

# Datalore Jetbrains

Installation using Docker or Kubernetes : <https://www.jetbrains.com/datalore/features/installation-and-configuration/>

Supports SSO, Docker image, Custom resource usage based on user

Features : <https://www.jetbrains.com/datalore/enterprise/>

Quickstart : <https://www.jetbrains.com/help/datalore/datalore-quickstart.html>

Latest feature releases : <https://blog.jetbrains.com/datalore/2022/06/09/datalore-enterprise-2022-2-blazing-fast-docker-based-installation-scheduled-runs-collaboration-on-attached-files-and-more/>

On premise licence plan , includes free for 1-4 users : <https://www.jetbrains.com/datalore/buy/?license=commercial>

Installing on premise : <https://www.jetbrains.com/help/datalore/datalore-enterprise.html>

Python, R, Scala Kernels : <https://www.jetbrains.com/help/datalore/kernel-management.html>

Allows Scheduling and machine management : <https://www.jetbrains.com/help/datalore/machine-management.html#manage-all-running-machines>

Datalore plugin (doesn’t seem to work well) : <https://plugins.jetbrains.com/plugin/12174-datalore>

Data Tools for Intellij : <https://www.jetbrains.com/data-tools/>

Whats difference between datalore & dataspell?

List of other tools and comparison : <https://datasciencenotebook.org/compare/zeppelin/datalore>

# Open Source Jupyter Notebook Editors

## Elyra

Almost everything that CDSW provides , Graphical pipeline building, Airflow/Kubeflow Integration, Spark/Kubernetes/Openshift support, autocomplete/syntax highlight etc IDE features, git & code reusability.

<https://elyra.readthedocs.io/en/latest/getting_started/overview.html>

<https://github.com/elyra-ai/elyra>

AI Pipelines with Elyra & Kubeflow : <https://www.youtube.com/watch?v=KR_m20pFXtU&t=33s>

## BeakerX

<http://beakerx.com/index>

BeakerX is a collection of kernels and extensions to the Jupyter interactive computing environment. It provides JVM support, Spark cluster support, polyglot programming, interactive plots, tables, forms, publishing, and more.

## Open Datahub

Tools ecosystem to run on openshift including Jupyterhub

<https://opendatahub.io/docs.html>

<http://opendatahub.io/landscape/>

<https://ruslanmv.com/blog/Docker-Container-with-Pyspark-and-Jupyter-and-Elyra>

# LF Open Source AI & Data foundation Landscape

<https://landscape.lfai.foundation/>

# Jupyter

All the extensions & other software related to Jupyter : <https://wiki.nikiv.dev/programming/interactive-computing/jupyter-notebooks>

## Spark Integration

<https://github.com/jupyter-server/enterprise_gateway>

<https://jupyter-enterprise-gateway.readthedocs.io/en/latest/>

Web server providing headless access to Jupyter Kernels. Provides support for Spark managed by Yarn, Kubernetes

Jupyter deployment & application server : <https://docs.jupyter.org/en/latest/projects/deployment.html>

<https://jupyter-enterprise-gateway.readthedocs.io/en/latest/>

Difference with enterprise gateway

you can create a multi-user Hub that spawns, manages, and proxies multiple instances of the single-user Jupyter notebook server.

Jupyter Enterprise Gateway does not manage multiple Jupyter Notebook deployments, for that you should use JupyterHub.

Not sure it supports spark?

# Other

## Jupyetr and Kernel Integration

<https://ipython.org/ipython-doc/3/development/index.html>

<https://ipython.org/ipython-doc/3/development/kernels.html>

<https://ipython.org/ipython-doc/3/development/messaging.html>

Ipython developer guide : <https://ipython.org/ipython-doc/3/development/index.html>

Is a program that runs & interprets users code. Ipython includes kernel for python

Ipython starts kernel & pass connection file to frontend specifying how to setup communication

Writing Kernel

1. Using iPython machinery/server to handle communication, just describe how to execute commands.
2. Implement your own kernel in machinery in target language

Kernel starts event loop, listening on heartbeat, control & shell sockets

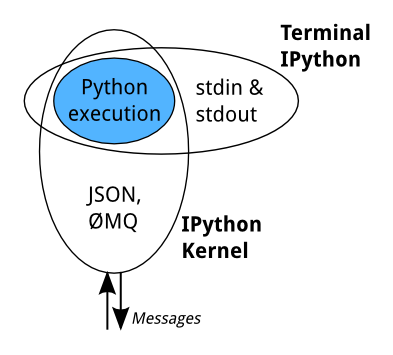
Hertbeat socket is used by frontend to check if kernel is alive

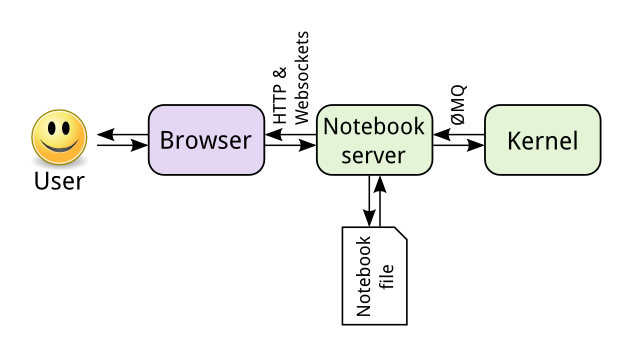
Messages in control & shell sockets?

Kernel will send messages on iopub socket to display output, stdin socket to prompt the user for textual input.

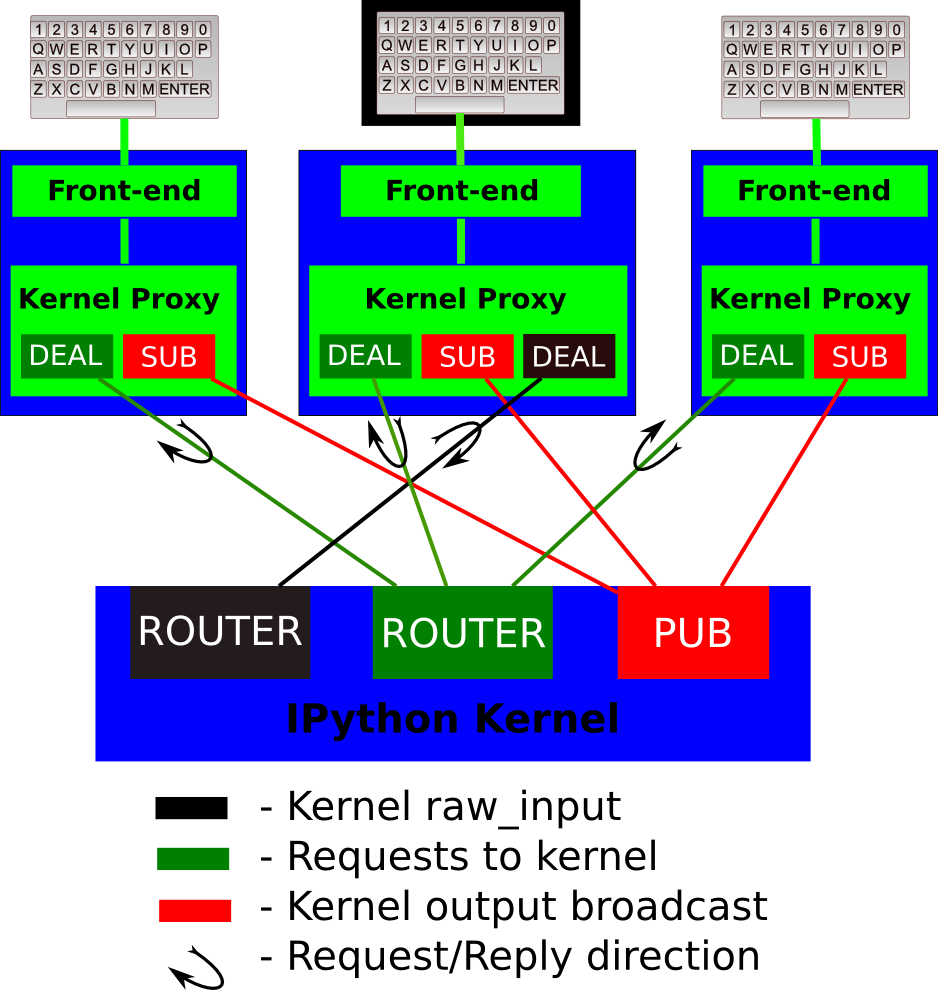
This is a separate process which is responsible for running user code, and things like computing possible completions. Frontends communicate with it using JSON messages sent over [ZeroMQ](http://zeromq.org/) sockets;

The core execution machinery for the kernel is shared with terminal IPython:



The notebook server, not the kernel, is responsible for saving and loading notebooks, so you can edit notebooks even if you don’t have the kernel for that language—you just won’t be able to run code. The kernel doesn’t know anything about the notebook document: it just gets sent cells of code to execute when the user runs them

the basic communications design and messaging specification for how the various IPython objects interact over a network transport. The current implementation uses the [ZeroMQ](http://zeromq.org/) library for messaging within and between hosts. The basic design is explained in the following diagram:



A single kernel can be simultaneously connected to one or more frontends. The kernel has three sockets that serve the following functions:

1. Shell: this single ROUTER socket allows multiple incoming connections from frontends, and this is the socket where requests for code execution, object information, prompts, etc. are made to the kernel by any frontend. The communication on this socket is a sequence of request/reply actions from each frontend and the kernel.
2. IOPub: this socket is the ‘broadcast channel’ where the kernel publishes all side effects (stdout, stderr, etc.) as well as the requests coming from any client over the shell socket and its own requests on the stdin socket. There are a number of actions in Python which generate side effects: [print()](http://docs.python.org/2/library/functions.html#print) writes to sys.stdout, errors generate tracebacks, etc. Additionally, in a multi-client scenario, we want all frontends to be able to know what each other has sent to the kernel (this can be useful in collaborative scenarios, for example). This socket allows both side effects and the information about communications taking place with one client over the shell channel to be made available to all clients in a uniform manner.
3. stdin: this ROUTER socket is connected to all frontends, and it allows the kernel to request input from the active frontend when [raw\_input()](http://docs.python.org/2/library/functions.html#raw_input) is called. The frontend that executed the code has a DEALER socket that acts as a ‘virtual keyboard’ for the kernel while this communication is happening (illustrated in the figure by the black outline around the central keyboard). In practice, frontends may display such kernel requests using a special input widget or otherwise indicating that the user is to type input for the kernel instead of normal commands in the frontend.  
   All messages are tagged with enough information (details below) for clients to know which messages come from their own interaction with the kernel and which ones are from other clients, so they can display each type appropriately.
4. Control: This channel is identical to Shell, but operates on a separate socket, to allow important messages to avoid queueing behind execution requests (e.g. shutdown or abort).

### Creating Simple Custom Python Wrapper Kernel

Have to extend **IPython.kernel.zmq.kernelbase , Implement methods**

**def do\_execute(self, code, silent, store\_history=True, user\_expressions=None)**

**do\_complete(*code*, *cusor\_pos*)**

**do\_inspect(*code*, *cusor\_pos*, *detail\_level=0*)**

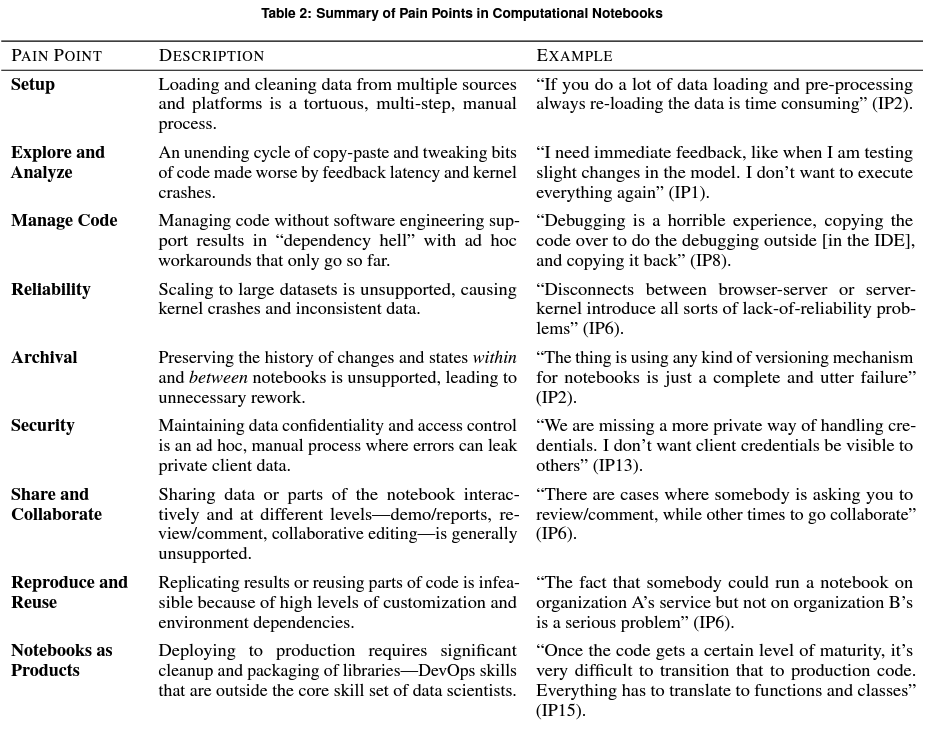
**do\_history(*hist\_access\_type*, *output*, *raw*, *session=None*, *start=None*, *stop=None*, *n=None*, *pattern=None*, *unique=False*)**

**do\_is\_complete(*code*)**

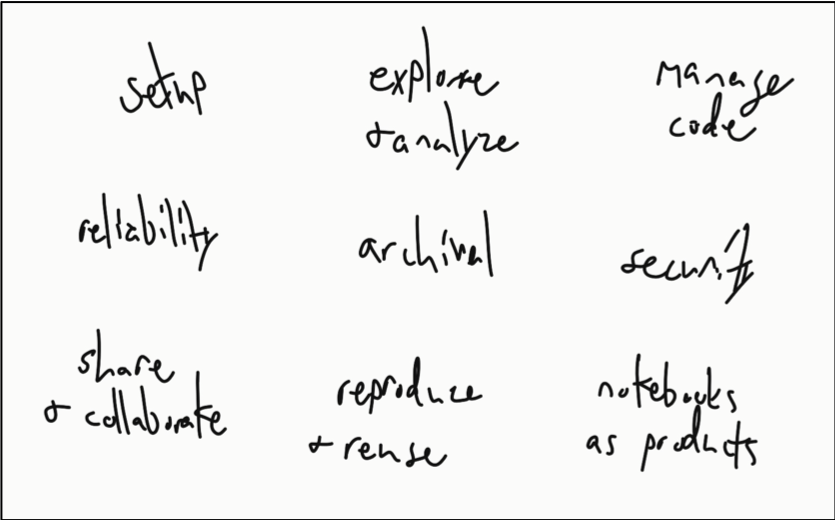
**do\_shutdown(*restart*)**

## Pain Points of Notebooks

What’s Wrong with Computational Notebooks Pain Points, Needs, and Design Opportunities : [Paper](https://drive.google.com/file/d/1tvtkHzCIpWGg8f-o0v8eCPCqEjszGmAg/view?usp=sharing) , [Notes on Paper](https://austinhenley.com/blog/notebookpainpoints.html) [Hackerrank discussion](https://news.ycombinator.com/item?id=22164916)



### Findings: 9 pain points

****

We identified the following 9 categories of painpoints based on our observations and interviews:

* **Setup.** Participants stated they often downloaded data outside of the notebook from various data sources since interfacing with them programmatically was too much hassle. Not only that, but notebooks often crash with large data sets (possibly due to the notebooks running in a web browser). Once the data is loaded, it then has to be cleaned, which participants complained is a repetitive and time consuming task that involves copying and pasting code from their personal "library" of commonly used functions.
* **Explore and analyze.** Modeling and visualizing data are common tasks but can become frustrating. For example, we observed one participant tweak the parameters of a plot more than 20 times in less than 5 minutes. Moreover, building models break the quick and iterative workflow of notebooks since it can take several minutes or longer to finish.
* **Manage code.** Notebooks do not have all of the features of an IDE, like integrated documentation or sophisticated autocomplete, so participants often switch back and forth between an IDE (e.g., VS Code) and their notebook. One participant we observed kept both windows side-by-side and copy and pasted code between the two windows rapidly as they worked. Another major pain point is managing package dependencies. Participants also indicated that they develop their own processes for debugging and testing, and some expressed irritation with the lack of tool support.
* **Reliability.** It is not uncommon for a notebook's kernel to crash in the middle of an operation, which may leave the notebook or data in an inconsistent state without proper feedback to the user. Participants commented that they find it easier to just restart and run the entire notebook again with hopes that it doesn't crash. Additionally, notebooks have limitations when it comes to big data, which requires users to move to a different tool set (e.g., Java or Python scripts).
* **Archival.** Participants expressed much difficulty with using version control systems for notebooks. For example, the outputs are saved in the notebooks along with metadata, which will always indicate changes to the version control system. Searching and finding information from previous notebooks is also an unsolved challenge.
* **Security.** Participants were concerned about sensitive data that may need to be masked from other users while still allowing them to execute the notebook. Notebooks also don't support restrictions such as read-only or run-only, thus requiring external tools to enforce access.
* **Share and collaborate.** While it is easy to share the notebook file, it is often not easy to share the data. For example, the data may require access to a database. Participants said that they often need to create documentation to explain how to install and setup any necessary dependencies to run a notebook. Furthermore, there is missing support for sharing the notebook results with others, especially non-technical users, for the purposes of reports or presentations.
* **Reproduce and reuse.** Due to the dependency issues and environment settings, it is unlikely that a notebook will work out of the box. Reusing even small portions of a notebook is difficult due to package dependencies and even dependencies on other cells within the notebook.
* **Notebooks as products.** If a large data set is used, as one might expect in production, then the notebook will lose the interactivity while it is executing. Also, notebooks encourage "quick and dirty" code that may require rewriting before it is production quality. For example, participants indicated that notebooks are not always designed to be executed top to bottom, which will require additional work to fix the execution order for a standalone artifact.

### Opportunities for Tools

Our findings highlight numerous opportunities for tools. From my own observations and conversations with data scientists, I think there are three major areas that tools should support:

* **Traditional development tools.** Notebooks are missing features that traditional IDEs have, such as autocomplete, documentation, debugging, unit testing, and refactoring. We observed participants repeatedly moving between tools to utilize these different features. Should IDE features be moved into notebooks or should notebooks be moved into IDEs?
* **Cleanup and extraction.** There are opportunities for tools to aid in cleaning up notebooks before archiving, sharing, or productizing. Since a lot of notebooks are started for exploratory purposes, it can be a lot of work to clean them up or to extract specific portions.
* **Feedback of notebook state.** Notebooks could provide more feedback to the user. What is the current state of the notebook? Which cells are dependent on each other? Which cells should be re-run?