Open Access | https://doi.org/10.25080/NRPV2311

SciPy 2024

July 8 - July 14, 2024

Proceedings of the 23nd Python in Science Conference
ISSN: 2575-9752

Any notebook served: authoring and sharing reusable interactive widgets

Trevor Manz[1], **Nils Gehlenborg**[1], and **Nezar Abdennur**[2,3]

[1]Department of Biomedical Informatics, Harvard Medical School, Boston, MA, USA, [2]Department of Genomics and Computational Biology, UMass Chan Medical School, Worcester, MA, USA, [3]Department of Systems Biology, UMass Chan Medical School, Worcester, MA, USA

Abstract

The open-source Jupyter project has fostered a robust ecosystem around notebook-based computing, resulting in diverse Jupyter-compatible platforms (e.g., JupyterLab, Google Colab, VS Code). Jupyter Widgets extend these environments with custom visualizations and interactive elements that communicate directly with user code and data. While this bidirectional communication makes the widget system powerful, its architecture is currently tightly coupled to platforms. As a result, widgets are complex and error-prone to author and distribute, limiting the potential of the wider widget ecosystem. Here we describe the motivation and approach behind the *anywidget* project, a specification and toolset for portable and reusable web-based widgets in interactive computing environments. It ensures cross-platform compatibility by using the web browser's built-in module system to load these modules from the notebook kernel. This design simplifies widget authorship and distribution, enables rapid prototyping, and lowers the barrier to entry for newcomers. Anywidget is compatible with not just Jupyter-compatible platforms but any web-based notebook platform or authoring environment and is already adopted by other projects. Its adoption has sparked a widget renaissance, improving reusability, interoperability, and making interactive computing more accessible.

Keywords Computational Notebooks, Jupyter, Widgets, Python, JavaScript, Data Visualization, Interactive Computing

1. Introduction

Computational notebooks combine live code, equations, prose, visualizations, and other media within a single environment. The Jupyter project [1], [2] has been instrumental the success of notebooks, which have become the tool of choice for interactive computing in data science, research, and education. Key to Jupyter's widespread adoption is its modular architecture and standardization of interacting components, which have fostered an extensive ecosystem of tools that reuse these elements. For example, the programs responsible for executing code written in notebooks, called **kernels**, can be implemented by following the Jupyter Messaging Protocol [3]. This design allows users to install kernels for various different languages and types of computation. Similarly, Jupyter's open-standard notebook format (`.ipynb`) ensures that notebooks can be shared and interpreted across different platforms [4].

Jupyter's modular architecture has also supported innovation in **notebook front ends** — the user interfaces (UIs) for editing and executing code, as well as inspecting kernel outputs. The success of the classic Jupyter Notebook [1] spurred the development of several similar Jupyter-compatible platforms (JCPs), such as JupyterLab, Google Colab, and Visual Studio Code. These platforms provide unique UIs and editing features while reusing Jupyter's other

Published Jul 10, 2024

Correspondence to
Trevor Manz
trevor_manz@g.harvard.edu

standardized components. This interoperability allows users to choose the platform that best suits their needs, while retaining a familiar interactive computing experience with the ability to share notebooks. Furthermore, the separation of computation from UI offers users a wide selection of both front ends and kernels. However, the proliferation of JCPs has led to significant challenges for Jupyter Widgets, a key component of interactive user interfaces in Jupyter.

Jupyter Widgets extend notebook outputs with interactive views and controls for objects residing in the kernel [5]. For instance, the ipywidgets library, besides defining the widget communication protocol, provides basic form elements like buttons, sliders, and dropdowns to adjust individual variables. Other community projects offer interactive visualizations for domain-specific needs, such as 3D volume rendering (ipyvolume), biological data exploration [6], [7], [8], and mapping (ipyleaflet, pydeck, lonboard), which users can update by executing other code cells or interact with in the UI to update properties in the kernel.

Widgets are unique among Jupyter components in that they consist of two separate programs — kernel-side code and front-end code — that communicate directly via custom messages Figure 1, rather than through a mediating Jupyter process. With widgets, communication is bidirectional: a kernel action (e.g., the execution of a notebook cell) can update the UI, such as causing a slider to move, while a user interaction (e.g., dragging a slider), can drive changes in the kernel, like updating a variable. This two-way communication distinguishes widgets from other interactive elements in notebook outputs, such as HTML displays, which cannot communicate back and forth with the kernel.

Widgets are intended to be pluggable components, similar to kernels. However, only the protocol for communication between kernel and front-end widget code, known as the Jupyter Widgets Message Protocol, is standardized. Critical components, such as the distribution format for front-end modules and methods for discovering, loading, and executing these modules, remain unspecified. As a result, JCPs have adopted diverse third-party module formats, installation procedures, and execution models to support widgets. These inconsistencies place the onus on widget authors to ensure cross-JCP compatibility.

Note

In this paper, we define a Jupyter Widget as consisting of a pair of programs: kernel-side and front-end, which communicate via the Jupyter Widget Message Protocol. While Python-only widget classes exist, which wrap other Jupyter Widgets as dependencies,

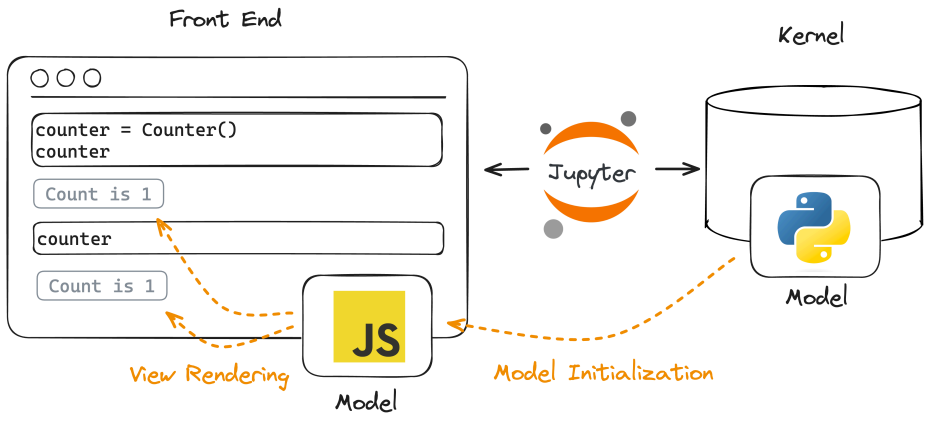

Figure 1. *Jupyter Widget conceptual architecture.*

> this paper focuses on those with dedicated front-end code, where the challenges described apply.

JCPs load front-end widget code by searching in various external sources, such as local file systems or Content Distribution Networks (CDNs) while kernel-side (Python) code loads and runs in the kernel Figure 2. These access patterns split the distribution of custom widgets between Python and JavaScript package registries, complicating releases and requiring widget authors to understand both packaging ecosystems. This division creates challenges, especially in shared, multi-user environments like JupyterHub. Since kernels cannot host static assets, users cannot independently install widgets or manage versions. Instead, widget front-end code must be pre-installed on the Jupyter webserver, typically by an administrator. Consequently, users are restricted to administrator-installed widgets and versions, unable to upgrade or add new ones independently.

These limitations make widget development complex and time-consuming, demanding expertise in multiple domains. They make user experiences across JCPs frustrating and unreliable. The high barrier to entry discourages new developers and domain scientists from contributing to widgets, limiting growth and diversity in the ecosystem. This leaves a small group of authors responsible for adapting their code for cross-JCP compatibility, hindering widget reliability and maintainability.

2. Methodology

The **anywidget** project simplifies the authoring, sharing, and distribution of Jupyter Widgets by (i) introducing a standard for widget front-end code based on the web browser's

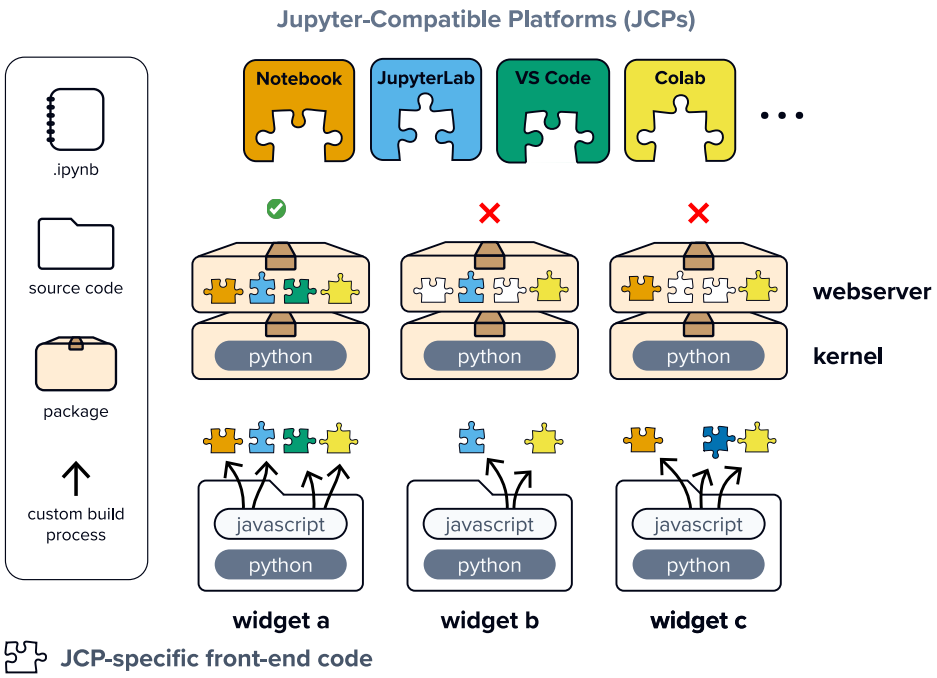

Figure 2. *Without anywidget, to ensure compatibility, authors must transform their JavaScript code for each JCP. Since JCPs load front-end widget code from a webserver rather than the kernel, widget front-end and Python code must also be packaged separately and installed consistently on any given platform. Missing puzzle pieces represent missing front-end extensions for specific target JCPs. The misshapen blue piece represents an incorrectly built or broken extension. Both situations contribute to fragmented JCP support.*

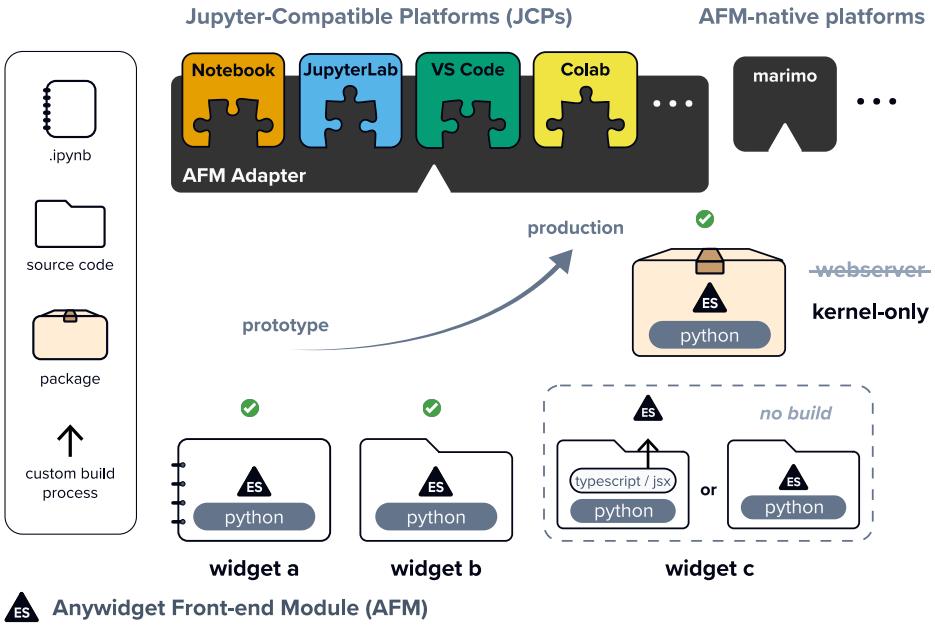

Figure 3. *With anywidget, developers author a single, standard portable ES module (AFM), which is loaded from the kernel and executed using the browser's native module system. For existing JCPs, anywidget provides a front-end adapter to load and execute these standardized modules, while new platforms can add native AFM support directly. Widget kernel-side code and AFM can be run directly from within notebooks, from source files, or distributed as single Python packages. With anywidget, each JCP consumes the same AFM, and widget authors can compose an AFM directly or target AFM through a build step to take advantage of advanced tools.*

native module system, (ii) loading these modules from the kernel, and (iii) providing the necessary "glue code" to adapt existing JCPs to load and execute these components Figure 3. This separation of concerns allows widget authors to write portable code that runs consistently across JCPs without manual installation steps.

Packaging custom Jupyter Widgets is complex due to the need to adapt JavaScript source code for various module systems used by JCPs. Initially, JavaScript lacked a built-in module system, leading JCPs to adopt diverse third-party solutions. Without a standardized widget front-end format, authors transform their code for each JCP. In the context of Jupyter Notebook and JupyterLab, this problem is described in the Jupyter Widgets documentation [9] as follows:

> *Because the API of any given widget must exist in the kernel, the kernel is the natural place for widgets to be installed. However, kernels, as of now, don't host static assets. Instead, static assets are hosted by the webserver, which is the entity that sits between the kernel and the front-end. This is a problem because it means widgets have components that need to be installed both in the webserver and the kernel. The kernel components are easy to install, because you can rely on the language's built-in tools. The static assets for the webserver complicate things, because an extra step is required to let the webserver know where the assets are.*

ECMAScript (ES) modules, introduced in 2015, are an official standard for packaging JavaScript code for reuse [10]. While most JCPs predate its standardization, ES modules are universally supported by browsers today. By adopting ES modules, anywidget is able to use the browser's native import mechanism to load and execute widget front-end code from the Jupyter kernel, thereby bypassing those used by JCPs and eliminating third-party dependencies. This approach not only overcomes many development challenges, but also

```
export default {
  initialize({ model }) {
    // Set up shared state or event handlers.
    return () => {
      // Optional: Called when the widget is destroyed.
    }
  },
  render({ model, el }) {
    // Render the widget's view into the el HTMLElement.
    return () => {
      // Optional: Called when the view is destroyed.
    }
  }
}
```

Figure 4. *An anywidget front-end module (AFM) with initialization and rendering lifecycle methods. For familiarity, AFM methods use naming conventions from traditional Jupyter Widgets; however, AFM narrows down the APIs provided to these methods, making it easier to load and execute AFMs in new environments.* Methods*: The* `initialize` *and* `render` *methods correspond to different stages in the widget's lifecycle. During model initialization, a front-end model is created and synchronized with the kernel. In the rendering stage, each notebook cell displaying a widget object renders an independent view based on the synced model state.* Arguments*: The interface of* `model` *is restricted to a minimal set of methods for communicating with the kernel (retrieving, updating, and responding to value changes). The* `el` *argument is a standard web* `HTMLElement`*.*

eliminates installation procedures for front-end code. Consequently, developers can prototype and share widgets directly within notebooks, making them more reliable and easier to use.

An anywidget front-end module (AFM) is an ES module with a `default` export defining widget behavior. This export includes lifecycle methods, or "hooks," for managing a widget's lifecycle stages: initialization, rendering, and destruction Figure 4. AFM lifecycle methods receive the interfaces required for kernel communication and notebook output modifications as arguments, rather than creating them internally or relying on global variables provided by the JCP. This practice, known as dependency injection [11], improves AFM portability by making integration interfaces explicit. New runtimes can support AFMs by implementing the required APIs, and existing JCPs can refactor their internals without breaking existing (any)widgets. For projects that want to take advantage of advanced front-end tooling, anywidget also provides authoring utilities to write AFMs such as "bridge" libraries for popular web frameworks [12].

Widget authorship is particularly challenging due to the need to integrate front-end code that communicates with kernel code in a heterogeneous set of environments. The anywidget project addresses these challenges by focusing on the standardization, development, and distribution of widget front-end modules, including the associated API to communicate with a computational kernel. The anywidget Python package serves as an adapter library that turns each JCP into an AFM-compatible host environment. Finally, the anywidget project provides additional tooling to help developers evolve their prototypes into mature packages [12].

An additional goal for an interactive computing paradigm like widgets is composability. For example, ipywidgets provides utilities to link widgets together or lay out grids of widgets on the page. These primitives allow widgets to derive from others, enabling reuse of front-end and kernel integrations in new ways. Importantly, Jupyter Widgets authored with the anywidget Python package extend from ipywidgets, making them interoperable with core ipywidgets and traditional custom widgets. By aligning with ipywidgets, anywidget promotes composability and prevents fragmentation of the ecosystem.

3. Features

Adhering to predictable standards benefits both developers and end users in many other ways beyond JCP interoperability, such as…

3.1. Web Over Libraries

Front-end libraries change rapidly and often introduce breaking changes, whereas the web platform remains more backward-compatible. Traditional Jupyter Widgets require extensions from UI libraries provided by JCPs, coupling widget implementations to third-party frameworks. In contrast, AFM defines a minimal set of essential interfaces focused on (1) communicating with the kernel and (2) modifying notebook output cells, without dictating models for state or UI. It allows widgets to be defined without dependencies, reducing boilerplate and preventing lock-in. Authors may still use third-party JavaScript libraries or tooling, but these are no longer necessary for JCP compatibility, publishing, or user installation.

3.2. Rapid Iteration

The web ecosystem's adoption of ES modules has led to new technologies that enhance developer experience and enable rapid prototyping. One such innovation is hot module replacement (HMR), a method that uses the browser's module graph to dynamically update applications without reloading the page or losing state. Since traditional Jupyter Widgets rely on legacy module systems, they cannot benefit from HMR and instead require full page clearance, reload, and re-execution to see changes during development. By contrast, anywidget is able to provide opt-in HMR, implemented through the Jupyter messaging protocol, in order to support live development of custom widgets without any front-end tooling. For example, adjusting a widget's appearance, such as a chart's color scheme, updates the view instantly without re-executing cells or refreshing the page.

3.3. Progressive Development

Anywidget makes it possible to prototype widgets directly within a notebook since all widget code is loaded from the kernel. Custom widgets can start as a few code cells and transition to separate files, gradually evolving into standalone scripts or packages – just like kernel-side programs Figure 3. In contrast, developing traditional Jupyter Widgets is a cumbersome process limited to the Jupyter Notebook and JupyterLab platforms. It involves using a project generator [13], [14] to bootstrap a project with over 50 files, creating and installing a local Python package, bundling JavaScript code, and manually linking build outputs to install extensions [15]. By removing these barriers, anywidget accelerates development and allows prototypes to grow into robust tools over time. In fact, as of January 2024, the JavaScript cookiecutter [13] project has been deprecated and directs users to use anywidget instead.

3.4. Simplified Publishing

Serving AFMs and other static assets from the kernel removes the need to publish widget kernel-side and front-end code separately and coordinate their releases. For example, many JCPs retrieve traditional widget Javascript code from npm, misusing the registry for distributing specialized programs rather than reusable JavaScript modules. Instead, with anywidget, developers can publish a widget (kernel-side module, AFM, and stylesheets) as a unified package to the distribution channels relevant to the kernel language, such as the Python Package Index. Consolidating the distribution process this way greatly simplifies publishing and discovery.

4. Impact and Outlook

Anywidget fills in the specification gaps for Jupyter Widgets by embracing open standards and carefully separating developer concerns. It defines an API for authoring portable and reusable widget components that decouples widget authorship from JCPs, resulting in multiple downstream benefits. First, anywidget—not widget authors—ensures compatibility and interoperability across existing JCPs, and authors can focus on important features rather than wrestle with build configuration and tooling. Anywidget's adapter layer simply loads and executes standardized front-end modules, meaning that the compatibility afforded does not introduce overhead to the runtime performance of a widget itself. A widget authored with anywidget has the same performance characteristics as a widget tediously (and correctly) authored using the traditional Jupyter Widgets system. Second, by circumventing bespoke JCP import systems and loading web-standard ES modules from the kernel, anywidget does away with manual installation steps and delivers an improved developer experience during widget authorship. Third, anywidget unifies and simplifies widget distribution. Widgets can be prototyped and shared as notebooks, or mature into `pip`-installable packages and distributed like other tools in the Python data science ecosystem. End users benefit from standardization because widgets are easy to install and behave consistently across platforms.

Since its release, anywidget has led to a proliferation of widgets and a more diverse widget ecosystem Figure 5. New widgets range from educational tools for experimenting with toy datasets (e.g., DrawData) to high-performance data visualization libraries (e.g., Lonboard, Jupyter-Scatter [16], Mosaic [17]) and research projects enhancing notebook interactivity (e.g., Persist [18], cev [19]). Many of these tools use anywidget's binary data transport to enable efficient interactive visualization with minimal overhead by avoiding JSON serialization. Existing widget projects have also migrated to anywidget (e.g., higlass-python, ipyaladin) and other libraries have introduced or refactored existing widget functionality to use anywidget (e.g., Altair [20]) due to the simplified distribution and authoring capabilities.

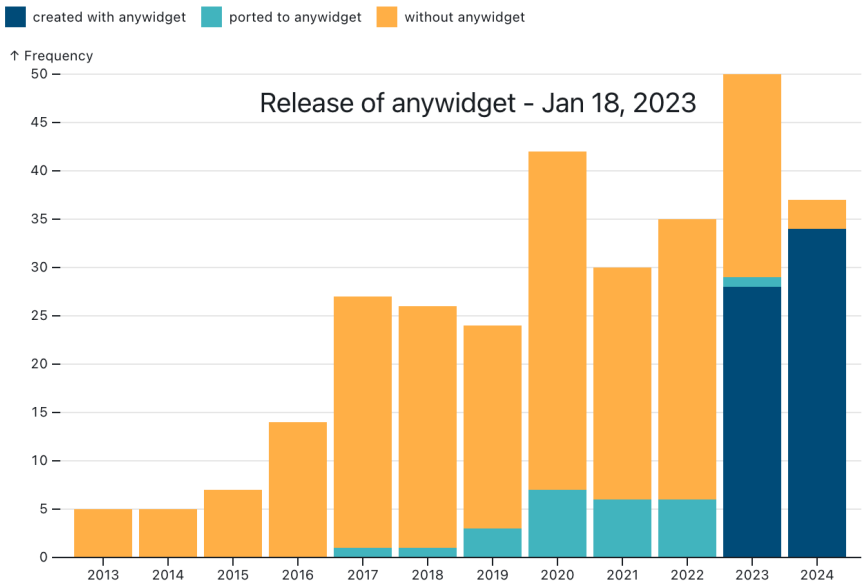

Figure 5. *Custom Jupyter Widgets per year as of September 2, 2024. Projects are tracked in a semi-automated process, combining daily code searches against GitHub with manual verification, at https://github.com/manzt/anywidget-usage. See repository for details and contribution guidelines.*

The portable widget standard also extends the anywidget ecosystem to platforms beyond Jupyter. Popular web frameworks and dashboarding libraries such as Voila, Panel, Shiny for Python, and Solara support Jupyter Widgets, and therefore also allow users to embed anywidgets in standalone web applications. Efforts are underway to add more specialized, built-in support for AFM as well. For example, marimo, a new reactive notebook for Python, is standardizing its third-party plugin API on AFM, allowing anywidgets to run natively without additional "glue code." Developers of the Panel web-application framework are also exploring deeper integration with AFM to enable reuse with their kernel-side reactivity systems.

AFM's standardization of widget front-end code extends reusability beyond the Python ecosystem. Its API structure is intentionally agnostic to backend processing, enabling creative and dynamic ways of hosting AFM-based widgets. Many simple widgets can function without any compute backend, while more complex ones can be upgraded with a computational backend when necessary. This flexibility allows for innovative hosting solutions across diverse platforms, from fully static web pages to full-stack web applications. By decoupling front-end interactivity from backend complexity, AFM empowers developers to create scalable, platform-independent scientific visualizations and tools, adaptable to a wide range of computational requirements and user scenarios.

A current limitation of anywidget is that using relative file imports for local dependencies requires bundling into a single ES module, which introduces a build step and prevents shipping AFM source code directly. This build step can be avoided by using a single file and loading third-party dependencies via URLs, though larger projects may still benefit from bundling. Notably, traditional widgets always require a build step and mandate using a specific bundler (Webpack) with a bespoke multi-JCP configuration. In contrast, anywidget makes bundling optional and targets a single standard ES module, which can be produced easily from a variety of bundlers. Future browser support for importmaps may enable local dependencies without bundling, simplifying anywidget development further as web standards evolve.

Tools for data visualization and interactivity have greater impact when compatible with more platforms, but achieving compatibility involves trade-offs [21]. The full capabilities of the widget system, such as bidirectional communication, are often inaccessible to authors due to development difficulty and maintenance efforts. Adopting standards can minimize these impediments, enabling both broad compatibility and advanced capabilities for users. A recent article enumerates these challenges and advocates for standardized solutions to democratize the creation of notebook visualization tools across notebook platforms [21]. Anywidget addresses this by introducing a standard that removes friction in widget development and sharing, making authorship practical and accessible.

Acknowledgements

We thank Talley Lambert for his contributions to the project, and David Kouril for his suggestions on the manuscript and figures. We also thank the anywidget community as well as members of the Abdennur and HIDIVE labs for helpful discussions.

Funding

TM, NG, and NA acknowledge funding from the National Institutes of Health (UM1 HG011536, OT2 OD033758, R33 CA263666, R01 HG011773).

References

[1] T. Kluyver *et al.*, "Jupyter Notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Scmidt, Eds., 2016, pp. 87–90. doi: 10.3233/978-1-61499-649-1-87.

[2] B. E. Granger and F. Pérez, "Jupyter: Thinking and Storytelling With Code and Data," *Comput. Sci. Eng.*, vol. 23, no. 2, pp. 7–14, 2021, doi: 10.1109/MCSE.2021.3059263.

[3] "Jupyter Client documentation." [Online]. Available: https://jupyter-client.readthedocs.io/en/stable/

[4] "nbformat documentation." [Online]. Available: https://nbformat.readthedocs.io/en/stable/format_description.html

[5] "Jupyter documentation." [Online]. Available: https://docs.jupyter.org/en/latest/projects/architecture/content-architecture.html

[6] T. Manz, S. L'Yi, and N. Gehlenborg, "Gos: a declarative library for interactive genomics visualization in Python," *Bioinformatics*, vol. 39, no. 1, p. btad50, 2023, doi: 10.1093/bioinformatics/btad050.

[7] T. Manz *et al.*, "Viv: multiscale visualization of high-resolution multiplexed bioimaging data on the web," *Nat. Methods*, vol. 19, no. 5, pp. 515–516, 2022, doi: 10.1038/s41592-022-01482-7.

[8] M. S. Keller, I. Gold, C. McCallum, T. Manz, P. V. Kharchenko, and N. Gehlenborg, "Vitessce: a framework for integrative visualization of multi-modal and spatially-resolved single-cell data," 2021. doi: 10.31219/osf.io/y8thv.

[9] "Low Level Widget Explanation." [Online]. Available: https://ipywidgets.readthedocs.io/en/stable/examples/Widget%20Low%20Level.html

[10] yu Shu-Guo, M. Ficarra, K. Gibbons, and E. community, "ECMAScript® Language Specification." [Online]. Available: https://262.ecma-international.org/14.0/

[11] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern." [Online]. Available: https://martinfowler.com/articles/injection.html

[12] T. Manz, N. Abdennur, and N. Gehlenborg, "anywidget: reusable widgets for interactive analysis and visualization in computational notebooks." OSF Preprints, 2024. doi: 10.31219/osf.io/tw9sg.

[13] "widget-cookiecutter." [Online]. Available: https://github.com/jupyter-widgets/widget-cookiecutter

[14] "widget-ts-cookiecutter." [Online]. Available: https://github.com/jupyter-widgets/widget-ts-cookiecutter

[15] "Anywidget documentation cookiecutter comparison." [Online]. Available: https://anywidget.dev/blog/introducing-anywidget/#a-solution-with-crumbs

[16] F. Lekschas and T. Manz, "Jupyter Scatter: Interactive Exploration of Large-Scale Datasets," *Journal of Open Source Software*, vol. 9, no. 101, p. 7059, 2024, doi: 10.21105/joss.07059.

[17] J. Heer and D. Moritz, "Mosaic: An Architecture for Scalable & Interoperable Data Views," *IEEE Trans. Vis. Comput. Graph.*, vol. 30, no. 1, pp. 436–446, 2024, doi: 10.1109/TVCG.2023.3327189.

[18] K. Gadhave, Z. Cutler, and A. Lex, "Persist: Persistent and reusable interactions in computational notebooks," 2023. doi: 10.31219/osf.io/9x8eq.

[19] T. Manz, F. Lekschas, E. Greene, G. Finak, and N. Gehlenborg, "A General Framework for Comparing Embedding Visualizations Across Class-Label Hierarchies," 2024, doi: 10.31219/osf.io/puxnf.

[20] J. VanderPlas *et al.*, "Altair: Interactive Statistical Visualizations for Python," *Journal of Open Source Software*, vol. 3, no. 32, p. 1057, 2018, doi: 10.21105/joss.01057.

[21] Z. J. Wang, D. Munechika, S. Lee, and D. H. Chau, "SuperNOVA: Design Strategies and Opportunities for Interactive Visualization in Computational Notebooks," in *Extended Abstracts of the 2024 CHI Conference on Human Factors in Computing Systems*, in CHI EA '24. New York, NY, USA, 2024, pp. 1–17. doi: 10.1145/3613905.3650848.