Benefits and Limitations of Jupyter-based Scientific Web Applications

Nicole Brewer* Rob Campbell†, Rajesh Kalyanam‡, I Luk Kim §, Carol X. Song¶, and Lan Zhao ‖
Research Computing, Purdue University
West Lafayette, IN, USA
*brewer36@purdue.edu, †rcampbel@purdue.edu, ‡rkalyana@purdue.edu,
§kim1634@purdue.edu, ¶cxsong@purdue.edu, ‖lanzhao@purdue.edu

2022 IEEE 18th International Conference on e-Science (e-Science) | 978-1-6654-6124-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/ESCIENCE55777.2022.00094

***Abstract*—Scientists are increasingly interested in creating standalone web-applications as computational and data analysis tools. The authors have worked with several such research groups to design, develop, and deploy such web applications that are increasingly based on Jupyter notebooks. One of the primary reasons among many to use Jupyter notebooks is the fact that research groups inheriting these applications are capable of maintaining and extending them. In this paper, we walk through the design process for one such application and discuss development environments that are best suited to Jupyter notebook development. We then explore several other applications where we employ similar design patterns. In doing so, we expound upon the benefits, limitations, and challenges of Notebook-based applications to provide a guide for other facilitators in similar situations.**

***Index Terms*—Jupyter Notebook, web applications, design, development, deployment**

I. Introduction

Scientists increasingly have the need to create web-applications that allow users to interact with their research in accessible ways. In the biological and environmental sciences, for example, the need has grown exponentially over the past decade. Web applications in these fields have been used for teaching, outreach, stakeholder engagement, and more recently, updating policy makers about the evolving state of the COVID-19 pandemic [1]. Additionally, web apps may augment scientific communication by providing a new medium of publication that is more useful and shareable than traditional formats.

In our Scientific Solutions Group, we have experienced demand for development in step with this trend. Researchers approach us with problems of varying complexity, and it is our job to analyze their requirements and work with them to establish which technologies and deployment strategies may best suit their needs. We have been developing scientific web apps and other tools that enable computational and data science. Recently, we have turned to Jupyter Notebooks for rapid development and deployment of these web applications. We have developed notebook-based web apps for five labs from a range of science, engineering, and social science domains. For example, we have developed several tab-based apps that progress through steps in the data life-cycle such as collection, management, analysis, and visualization. Such applications may promote reuse of valuable data sets. Other apps serve as advanced calculators where the graphical interface allows other researchers to specify input parameters and run a complex computation without ever interacting with the underlying code. These apps improve the accessibility of scientific software by obviating the need for advanced programming skills.

However, Jupyter is not a panacea for web application development; it has many limitations. Some of these drawbacks are obvious before development while others have only manifested while we were developing them. Some of these challenges have been ameliorated through design, while others have been resolved by deploying the app on cyberinfrastructure (CI) resources. Finally, some limitations serve to narrow the scope of the appropriate applications of Jupyter Notebooks. In this paper, we attempt to organize and highlight the individual differences among our applications to suggest when and how Jupyter Notebooks can or should be used to create web applications. This work is by no means a comprehensive recommendation, but rather a report on our experience that may serve as a guide to those who are considering developing a web application for themselves or for other research groups.

A. Audience

We communicate our findings from the perspective of a research software engineer (RSE) who is tasked to facilitate the development of a scientific web application that is owned by a research lab. We review our experience at depth and breadth so that it may be useful to researchers and RSE's alike.

The high level sections may be of interest to those who are developing or are considering a web application as a means to share their research. Researchers can use this paper to determine whether they need to employ the expertise of an RSE to accomplish their aims. Researchers with minimal requirements may be able to use our templates to independently kick-start their application development. However, common requirements such as shared storage, authenticated users, and future scalability may require additional expertise.

When requirements are more complex, researchers may need to collaborate with a facilitator such as an RSE. Because these web applications are becoming increasingly popular and important in research, it is imperative that RSE's understand the benefits and limitations of various web development tools

978-1-6654-6124-5/22/$31.00 ©2022 IEEE
DOI 10.1109/eScience55777.2022.00094

and design choices. In sub-subsections, we discuss more technical topics such as object-oriented design patterns and non-trivial deployments. These subsections are intended for an audience of our peers in research-facing or facilitation roles.

B. Scope

Our team has increasingly used Jupyter Notebooks [2] to deploy web apps in collaboration with researchers. These libraries comprise a diverse set of widgets - components that bind Python code to a graphical user interface running in the browser. Widgets, such as Dropdown and Textbox, allow users to interact with the underlying variables and functions without coming into direct contact with it. For example, moving a slider may change the value of an integer, or clicking a button may trigger a callback function that is responsible for running a computation or downloading a file.

Jupyter Notebooks support interactive programs in manifold languages by way of kernels [3]. A kernel, such as IPython, is a language specific process responsible for executing code cells when running a Jupyter Notebook. To the best of our knowledge, only three Jupyter kernels have dedicated interactive widget libraries that support object-oriented user interface development in Jupyter Notebooks [4] [5] [6].

Language	Kernel	Widget Library
Python	IPython	ipywidgets
C++	zeus-cling	xwidgets
Julia	IJulia.jl	Interact.jl

All of these widget libraries are designed to be built upon. Further, it is possible for anyone to build custom widgets. All widgets have a JavaScript front-end view that communicates bidirectionally with a Python back-end model. For example, the ipyleaflet [7] library simply ports functionality of the leaflet.js [8] JavaScript library with a Python back-end to create an interactive map widget. There are many open-source widgets that are well-maintained by the Jupyter Project and other organizations, so there is a rich ecosystem of libraries that extend the functionality of the core widget library. For example, bqplot is for 2-D plotting [9] and ipywebrct [10] is to capture audio, video, and other media from variety of sources.

All three of these widgets are also available in xeus-cling as xleaflet, xplot, and xwebrct, respectively [11] [12] [13]. These libraries use the same front end as their IPython counterparts, but the back-ends have been rewritten in C++. Many other well-maintained widgets, such as ipysheet, do not currently have equivalent counterparts available in C++.

Of these kernels, IPython is by far the most widely used [14]. One advantage of the ipywidgets framework is that it takes advantage of the IPython display library which is responsible for rich displays of many kinds of media and Python objects including lists, dictionaries, images, gifs, audio, video, and many others. Any object that can be displayed in IPython can also be displayed inside an Output widget. For example, pandas provides support for a rich display of DataFrames, which can be wrapped in an Output widget to be incorporated into a larger application. This is possible because ipywidgets supports composability through container widgets.

ipywidgets is not the only library that provides data-manipulating widgets; there are several Python libraries suitable for creating interactive data science dashboards. Libraries such as altair [15] and bokeh [16] achieve interactivity through many of the same widgets as ipywidgets. However these libraries are declarative, which means the syntax places emphasis on data visualization at the expense of object orientation. This design may be well suited for data dashboards, but we anticipate that these libraries would not allow for modularity sufficient enough to scale with new requirements as the application matures. For these reasons, our group has exclusively written these apps in Python with ipywidgets, so we limit the scope of this paper to that experience.

C. Benefits of Developing Notebook Applications

There are many benefits to creating web applications to share research, but there are also barriers to entry. Writing user interfaces in Python resolves many of the encumbrances associated with the traditional web development stack. It requires no prior knowledge of traditional web development, but as we will see, it is still easily deployed on the web. For an RSE, it is quick and easy to learn how to develop a user-interface in Python. For scientists who may inherit the app, all of the code is in one place. The entire web development stack is compressed into a Python library instead of an entire framework. Most of our web applications are comprised only of one Jupyter Notebook and three Python files. Maintenance doesn't require digging through a labyrinth of files and directories that accompany even the simplest web frameworks. Additionally, launching and interacting with the Jupyter Notebook server through the Anaconda Navigator reduces the need for command line interfaces that can be confounding to users without extensive experience in UNIX. Overall, development in Python is approachable to any researchers or domain experts who wish to take over maintenance of the project on completion of development.

D. Inherent Limitations

The benefits of easy development has trade-offs. Some of these are obvious at the start of design and serve to narrow the scope of appropriate application of Jupyter Notebooks. The cost of avoiding traditional web development is that developers have little stylistic control. While it is possible to use custom CSS to alter the look and feel of widgets, we find that drastic alterations defeat the simplicity of development. Because ipywidgets is built on top of JavaScript, there are many kinds of widgets available. The development team boasts of a rich ecosystem of third-party widgets, including the very successful ipyleaflet for interactive maps. Four of these are maintained by the ipywidgets team. There are many others written by individuals and small teams, however there is always a risk that a third-party widget will not be maintained in the future. Other limitations presented themselves at the

time of development, and those are the challenges we expound upon below.

II. Basic Implementation

The previously highlighted benefits of using Jupyter Notebooks in practice can be best evinced by presenting an oversimplified use case. Suppose a researcher approaches a RSE group with code in hand; a notebook that allows a user to filter and visualize a dataset. The researcher says they want to turn their code into a standalone web application that can be accessed by anyone with a link.

The RSE group is enthusiastic about this project because it will take a minimal number of development hours. All they need to do is add some widgets that accept a file for data upload and wrap the various input parameters with sliders and number selectors. This application is essentially a data visualization dashboard. The RSE creates an Anaconda environment file to the repository to capture a semi-reproducible execution environment for running the notebook. They add Voila [17] to the list of dependencies to enable launching the notebooks as a standalone application. At this point, the repository is ready to be hosted with Binder [18], a free service that allows users to deploy notebooks on the web. Many examples of this simplified use case can be found in the Voila Galley [19]. Because the application utilizes the researcher's core code and because the need to learn the standard web development stack is avoided, the researchers can maintain and extend the codebase on their own after the initial stages of development.

This example is meant to demonstrate the benefits of using Jupyter Notebooks for creating scientific web apps. Thus far, we have not come across a case simplistic enough that all the code cleanly fits into a single notebook. And invariably, our collaborators have requirements that necessitate deployment on cyberinfrastructure resources. However, if all the data for an application fit into a small csv file, it would be possible for a researcher to host the application online for free with Binder. This level of development might be achievable without the aide of an RSE.

1) Work with Tabular Data in Pandas: Researchers very often work with tabular data, which is data arranged in rows and columns and viewed in a spreadsheet. The pandas [20] library allows users to explore, clean, and process tabular data in a programmatic way instead of through the use of spreadsheets. It is an excellent way to work with data in IPython, because it has built-in functions that seamlessly display the data in a notebook. Hence a developer can integrate a DataFrame into an application by displaying it in an Output widget. These displays do not automatically reflect updates when changes are made to the DataFrame, but it is simple to refresh the visualization when the DataFrame has been updated.

Because of this - and because it is the standard for DataFrames in Python - pandas makes an excellent choice when dealing with tabular data in Jupyter-based applications. pandas makes it easy to read in a csv file for filtering or manipulation.

2) Interactively Select Data with bqplot: The next step in this simple application is to filter and select data. This can be done through the use of standard ipywidgets. Selecting a date can be accomplished by populating an IntSelect widget with all the years that span the range of the dataset.

This can also be achieved with the rich interactive visualizations provided bqplot. For instance, a researcher may have a dataset that describes all the patents in the biotechnology industry over the past 30 years. With bqplot, it is simple to create a 2D plot of the total number of patents awarded on the y-axis and the year on the x-axis. If you plot this data with a scatter plot, you can click and drag in order to visually select a range of years with what is called a BrushIntervalSelector.

bqplot provides a satisfying way to select and filter pandas DataFrames. However, we have found that this library is more well suited to data science applications than computation ones, which we will discuss in a later section. The last step in this hypothetical application is to allow users to download the resulting dataset. Once this is achieved it is ready to be deployed to the web.

3) Deployment with Binder: Now that this application is complete, it needs to be made available to anyone on the web. Binder [21] is a free resource that deploys notebooks in an executable environment so that the code cells can be rerun. Deploying with Binder has few requirements: the notebook needs to be housed in a repository available on the web, such as GitHub; and it needs to have a file that specifies your dependencies, such as requirements.txt or a environment.yml file. Binder does the hard work of creating a suitable Docker image behind the scenes.

If those steps are followed, Binder will open up JupterLab to the application notebook. However, it would be preferable if the users of the app weren't able to see the editor, or even the code cells that are responsible for rendering Widgets in the cell output. This is the purpose of Voila; it turns a single Jupyter Notebook into the standalone application just described. Voila does this by running the code in the notebook and only displaying the output such that codes cells are hidden.

Voila can be used in a couple of ways, but when deploying an app with Binder, the best way is to point Binder toward the JupyterLab server extension. This integration with Voila ships natively with JupyterLab, so the only step that needs to be taken to deploy a standalone application on Voila is to specify the "URL to open" option on the Binder configuration page to `voila/render/<application-notebook>.ipynb`.

And Voila!, Binder will deploy the standalone application that can be accessed by anyone on the web. This simplified example is much like our CoExplorer application, but these researchers had one additional requirement that precluded the deployment with Binder. The next section explores this and other challenges with development and deployment of Jupyter-based applications.

III. Challenges and Known Limitations

Next we present each application and requirements for it. Each new application brought a new dimension of functional-

ity that added to the complexity to either the development or deployment of the application. As we walk though each use case we will address how we attempted to resolve each new challenge.

A. CoExplorer

Researchers within Wisecaver Lab at Purdue University required a standalone web application that would allow other genomics researchers to query and visualize their gene coexpression network data. For selected plant species, they had developed Python code to visualize results of experiments on gene coexpression networks for a set of stressors (light, water, heat, pathogens, etc.). These researchers wanted to transform their investigatory code housed in Jupyter notebooks into a usable app with a polished user interface. Further, they wanted to be able to maintain and extend features of the app themselves, without needing to delve into traditional web app development (i.e. HTML, CSS, JavaScript, etc.) or other frameworks. They also wanted to be able to clone and revise the codebase in order to make applications with similar functionalities but specialized for different data sets.

They had one additional requirement that distinguished them from the simplified use case above: the ability to access and quickly search within sizeable data sets. This meant they needed our help to connect their app to a campus data depot.

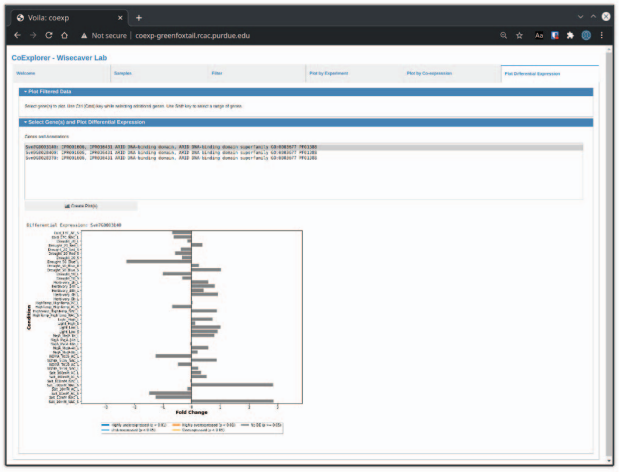

Fig. 1. CoExplorer was the original inspiration for the tab-based MVC template.

This collaboration typifies our ideal use-case for several reasons. The researchers had a detailed vision of what they wanted the app to look like, they came to us with some exploratory code that made the project easy to start from, and they wanted to maintain the app themselves after some initial development.

Because of the success of this use-case, we decided to create a template out of this application in order to facilitate code reuse and cut down the development time on similar projects [22]. The application contained enough code that we ported all of the cell-based code into three Python files that were organized by a traditional Model-View-Controller (MVC) pattern. With the help of this template, the lab was able to take over the development of their app and create a suite of similar applications. This template is free and open source for use by any researchers or RSE's who are taking on a similar project.

1) Incorporating Common Plotting Libraries: This CoExplorer application (Fig.1) used both the matplotlib and plotly libraries to produce interactive data visualization features. In particular, plotly visualizations were configured to produce graphical coexpression networks that can be zoomed and rotated in three dimensions. This was possible because plotly has a FigureWidget that is a graph object that extends ipywidgets. Matplotlib was incorporated into the ipywidgets ecosystem with the ipympl [23] library, which we will revisit in further detail.

2) Shared Storage: This template is designed with deployment in mind. As with the simplified use case, the template uses Anaconda, Voila, and Docker for reproducible deployment. However they had one requirement that differentiated their use case. The app required shared access large data sets that needed to be accessed from shared storage. This common requirement is what excluded them from being able to use Binder to deploy their tool for free. Alternatively, we were able to host their application on the Purdue University Geddes platform [24], which uses resource managers such as Kubernetes to scalably deploy sessions of the application. Each instance of the application is deployed as a Docker container with access to the shared storage database.

B. AgMIP Explorer

AgMIP Explorer (Fig.2) is a data exploration application for the Agricultural Model Intercomparison and Improvement Project (AgMIP) GlobalEcon group that conducts systematic intercomparisons from predictive agricultural and socioeconomic models in order to improve predictions about the future performance of food systems. The international group investigates food crop related metrics and forecasts as reported and inferred by countries and regions across the globe.

In this project, multiple teams submit result data based on each team's model periodically, and therefore must produce data that either originally, or via scripted transformation, conforms to the project's standard format. Currently done through a manual process, they wanted us to build a web application that would allow them to query, compare, and visualize datasets generated from multiple models.

While the combined data was not prohibitively large, parts of it were updated frequently by various teams around the world. It made sense to us to create an additional tool to facilitate data upload, validation, and submission. We subsequently used the same template to create an additional Data Submission tool that allowed their teams to maintain the datasets used by the Data Explorer tool. The supplemental, "AgMIP GlobalEcon Data Submission Tool", was based on the same Jupyter-ipywidgets-MVC template. It let research teams upload their new data, examine it against the project's data

format standard, and adjust format and standard data values as needed. Users could then submit the data to be saved in a secure area prior to publication in the Explorer tool.

Because of this, they required controlled user access to the application. They needed regular user logins as well as users that were granted permission to update shared dataset. This is not a feature that can be easily incorporated into a deployment with Binder. We worked with them to deploy the app as a tool on a science gateway in order to give them the ability to administrate user groups. After an initial development phase, they were able to maintain the project themselves.

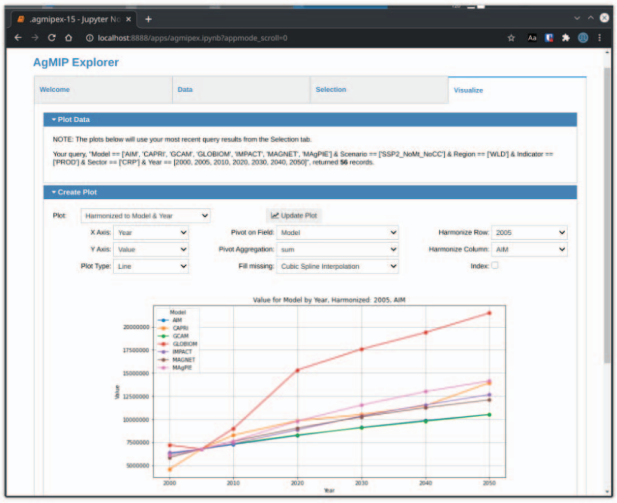

Fig. 2. AgMIP Explorer is another tab-based web application.

1) Authenticated Users: Because they needed user authentication and group management, we hosted the application as a tool on MyGeoHub [25]; a science gateway built with HUBzero [26]. Their datasets were too large to be hosted in a shared GitHub-hosted repository and needed to be housed in a shared data storage accessible by only authenticated users that belong to the tool group, which HUBzero handles very easily. We chose to initialize the app from our notebook template to facilitate quick development and easy maintenance. The tool employed pandas for plotting and Matplotlib for visualization.

C. Superpower

Superpower is an extensive R package by the Social Cognition of Social Justice lab designed to help psychologists perform power analysis on their study designs. We worked with the researchers to create a graphical user interface (GUI) for the package so their users can perform the calculations without the need to program. This application has proven to be the most complex application with the longest development time and the largest code base. As such, we have in many ways pushed up against the limitations of ipywidgets-based application. This application was designed with a Tab-based structure that can be seen in Fig.3.

It contained a large number of interdependent widgets, which necessitated a deliberately modular design that required extensive knowledge of object-oriented programming. Another challenging aspect of development was finding solutions for charts and graphs with interactive components such as a click-and-drag error bar. ipywidgets doesn't have an all-in-one solution for building these interactive visualizations, and it took many attempts to achieve a modest solution in many cases.

One challenge of point and click interfaces can be reproducing particular settings when an interesting outcome is found. Superpower was a complex application with many interdependent variables, for example, changing a variable might change the dimension of a matrix containing dynamic values. Even so, this complex application can still be considered a calculator, because every unique set of input parameters was supposed to invariably calculate the same output. In other words, a user may click through the app to set a unique set of parameters in any number of ways. As long as the parameters are the same, the result should always be reproducible.

This is an important, and often neglected feature of point and click interfaces. When a researcher obtains an interesting or useful result, anyone who reads and wishes to reproduce that outcome should be provided with all the information they need to do so. Superpower was designed with reproducibility in mind. For every calculation performed, the set of parameters that produced that result were also recorded. From the results page, a user would be able to repopulate all of the parameters and rerun the calculation to reproduce the same result. Results and their parameters were both exportable. This way, if a researcher were to share the results of the calculation in a publication, they could also share the settings used to produce it, making it reproducible by other researchers.

Superpower is a five-year project. Due to the complexity of this application, there has been an RSE responsible for the user interface for the entire duration. Researchers and developers wanting similar features may want to read on or consult with an RSE to determine if Jupyter Notebooks is an appropriate choice for their application.

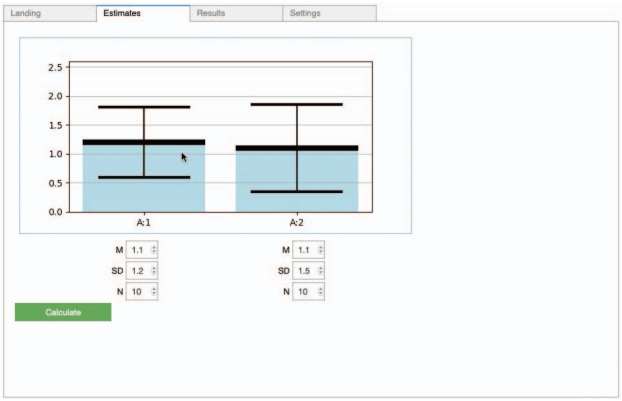

Fig. 3. Superpower is a tab-based web application that employs a modular ModelDelegate design pattern.

1) Modularity: One of the defining features of Superpower that influenced the initial design of the app was the need for components, which we will define here as a Container widget such as a Grid, Box, or Tab that contains one or more children. Components are an important feature in this application because these sets of widgets often needed to be repeated within multiple places in the application. For example, Fig.4 shows a Grid of FloatText and Buttons that was tightly coupled with a complex underlying model. The application required two instances of this component to appear in different parts of the application. In turn, we had to design this application in a modular, composable way.

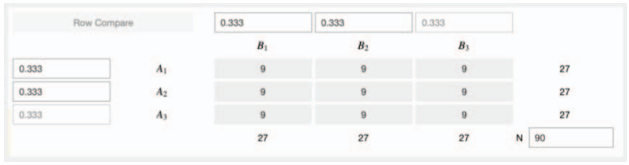

Fig. 4. A modular component that was reused throughout the app.

Instead of using our traditional MVC template as with other applications, we extended the ModelDelegate design pattern [27]. This is the design modeled by the ipywidgets library, where the View and the Controller make up an inextricable component called the User Interface Delegate. That is to say, the Views simultaneously display the state of the Model and manipulate it. To confirm this with a thought experiment, imagine a slider widget. As you drag the slide, you are simultaneously changing the view by moving the location of the slide handle and changing the value the slide represents. We extended this design pattern to compose the application in a modular way. For example, one of the tabs in the application was dedicated to user configuration. We developed a class that maintained the state of the user configuration, and updated the user's configuration file if one of these attributes changed. We separately created a View component that extended a Box widget and contained all the widgets needed to represent the state of the configuration file. We then extended the View class so that the Model was one of its attributes and it contained observe methods that would update the Model. This View extended with the features of a Controller is what we call a component.

The first advantage of this design is that the Components are composable in the same way that widgets are. You can have a Container widget inherit a Component in the same way it can inherit another widget. For example, a Tab widget can have a user configuration component as a child. The consequence of this is that the application can be written modularly. It is possible to develop and test the user configuration Component entirely on its own, without having to relaunch the entire application every time you update that part of the codebase.

One signature of the Superpower tool, compared to other existing tools for power analysis, is the highly interactive graphs beyond the usual parameter setting. For example, the science team wanted users to be able to click and drag error

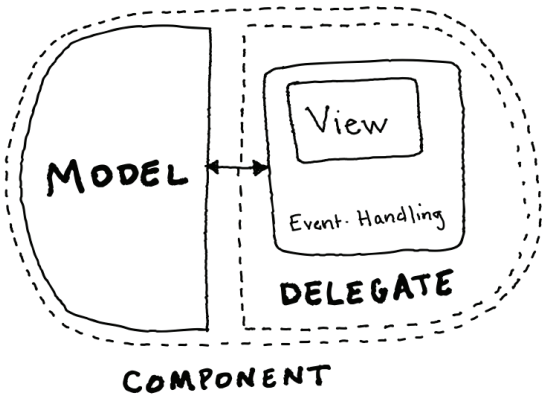

Fig. 5. A component is made up of a Model and a Delegate. A delegate is a Controller that encapsulates a View

bars directly within bar charts. As we will see, this requirement is more onerous to satisfy than say, controlling an error bar in a bar chart by clicking and dragging a mediating Slider widget. Using an interactive widget to update a visualization is unidirectional. It may be possible to click and drag the error bar from within the bar chart, but out-of-the-box there is no way for this change to be reflected back to the Slider widget. When building applications with ipywidgets, trait changes are only observable from a widget that is built on top of the traitlets [28] library.

While developing Superpower, our group has employed several strategies to create interactive plots that can communicate bidirectionally with the traitlets library. In our experience, there is not a single comprehensive strategy for achieving this. To the best of our knowledge, there is only one plotting library that is built on traitlets and a member of the ipywidgets ecosystem. In the bqplot library, "every single attribute of the plot is an interactive widget" [9]. This is advantageous because this means that changes to the plot can be observed by a widget that is not a member of that plot - achieving bidirectional trait communication.

This ostensibly sounds as though a developer can observe any component in the plot, but this is not the case. Trait notifications are available only for certain plot components. In the case of Superpower, this fact limited the usefulness of this library. For instance, the science team asked for an line plot where users would be able to pick and drag markers to change the endpoints of each line. This feature is not implemented in bqplot, which is surprising because markers in scatter plots do have `enable_move` and `restrict_y` traits that allow users to click and drag markers to change a scatter point coordinate along a single axis. Line plots do indeed have customizable markers that can take on many shapes and sizes, but these markers are not movable in the same way

that they are in scatter plots. There were several ways to hack a scatter plot and a line plot together to implement this feature, but each result significantly detracted from the user experience. It is limitations like these that keep bqplot from being a comprehensive solution to interactive plotting. This is especially true when it comes to manipulating data as opposed to selecting data, which we perceive to be well-supported.

2) Interactive Plots: Where these libraries do not support the level of interactivity needed, it may be necessary to use JavaScript. We used JavaScript when first developing interactive charts in the Superpower application. It was easy to add a JS library requirement and create a DOM element to be used for a plot but major limitations quickly arose. It was seemingly impossible to maintain a composable, object-oriented approach. This is because we were able to send data from Python to JavaScript, but not the other way around. Because of this limitation we refactored our code. But since then, a PyPi package called NotebookJS [29] has been developed to ameliorate the Model communication problem between Python and JS, though we don't have any experience using it.

We considered wrapping our JavaScript code in a custom widget, which is a feature provided by the ipywidgets ecosystem. Custom development is desirable because there are many well-maintained pure JavaScript libraries that are worth wrapping in Python. For example, ipyleaflet is a custom widget built underneath leaflet.js. This library has many of the features available to the JavaScript library, like zoom in, zoom out, pan, and add a marker. A custom ipywidget is composed of a JavaScript object on the frontend and a Python model on the backend. The Python backend shares attributes with the JavaScript frontend, so a developer can poll the widget for, say, the coordinates of a marker that was placed interactively or define a Python callback to observe updates to the location of a particular maker. However, this kind of development is non-trivial for developers who don't have experience in object-oriented JavaScript. The concept of context and the associated `this` keyword and `bind` function are nontrivial concepts. Custom widgets may not be worth the hassle for developers without JavaScript experience, but for those that do, this mechanism opens many doors.

We then turned to interactive Matplotlib (ipympl) which is a Matplotlib graph with pan/zoom, mouse-location tools, and other events [23]. Though interactive Matplotlib is very versatile and stylistically very customizable, we found this solution to be equally stultifying for the same reason as before: ipympl charts are easy to manipulate and update, the events aren't compatable with traitlets. Though we were able to achieve bidirectional communication between the interactive graph and a traitlets model in several ways, none of them were performant enough to be a viable solution. Specifically, it was simple to click (or "pick") a bar in a bar graph to change its color attribute. But clicking and dragging on the height of the bar in such a way that traitlet attributes belonging to the Model were also updated, was difficult. Small changes improved performance somewhat, but ipywidgets did not prove to be an ideal solution overall for this highly interactive use-case.

In summary, we have yet to find a good solution for bidirectional interactivity. There are many options for using ipywidgets to update a visualization, but we have yet to find a library suitable to reflect highly-interactive changes in the graph back to the Models they represent. This is presumably more important for applications that seek to manipulate data instead of simply cleaning or filtering it.

3) Dynamically-sized Widget Arrays: As shown in Fig.4 Superpower contained many components that represented multidimentional matrices. Perhaps the easiest example to visualize is the Grid components that contained a 2-D array shown below.

This component is highly interactive, meaning that user manipulations of any of the internal IntText widgets required a cascade of updates to the Model, which in turn needed to be reflected in the widgets on the perimeter. Unfortunately, the traitlets library doesn't support container traits such as arrays. This means that when an element in a list or numpy array is updated, it doesn't trigger an event.

One work-around would have been to not completely extricate the Model from the Delegate, and to combine the logic and the view together in a single component. This does not work for our use case because we needed to pass each Model through several Delegates, so concerns had to be purely separated. Separating concerns is best practice, especially if applications are expected to grow in complexity.

Another solution might have been to have many individual Int traits in the Model to represent each IntText in the UI. This might be reasonable depending on how many Grid elements their were. For example, you could represent a 2x2 grid with four Ints instead of a 2x2 numpy array. This wasn't an option for us, because the size of the grid was dynamic. A user could chose a 2x2 grid or a 3x5 grid so there was no way to hard code these integers.

Therefore, we needed to be able to observe a change in an array, which is something that traitlets doesn't support out of the box. One solution we found was to use the traits library [30] (as opposed to the traitlets library) to implement the Model [31]. Unlike traitlets, the traits library does support container traits. This means that a change to a single element in a list or array would be observed, as shown below. We perceive two downsides to this approach. One is that it requires the developer to become familiar with a syntax that is similar, but distinct from the library they are already familiar with. The second is that it bars the developer from using the ability to `link` Model traits with Delegate traits. While this functionality can be alternatively be achieved with one or two observe functions, using a `link` takes fewer lines of code and is likely more readable.

Our last and best solution was to use a functional programming library, functools [32] to apply an observe function to a dynamic number of the Grid children. Using the `partial` function, we were able to loop through assigning the same callback to each widget. The callback was modified so that it received the index the widget that called it. This was a

critical piece of information, because the objective of the callback function was, in-part to update the corresponding array attribute in the Model with the updated value. This could not be done without supplying the index of the Widget in the Grid. This was possible because `partial` enables assigning these function parameters at the time that it assigned the callback.

IV. Templates and Development Environments

We have found that a traditional development environment can be burdensome. Static files are edited, and the entire notebook has to be rerun for changes to take affect. If a developer is working on a component that is well-embedded in a complex application, it may take many click-throughs to observe the changes affected by the modified code. Several integrated development environments ameliorate these issues with various approaches.

Our group has developed all three of these apps within the past few years. Over time, it became salient that notebook-based template apps solve a common problem. We decided to take some up front development time to extricate common features into a template called nbtmpl [22]. This template employs a traditional MVC design pattern where source code is divided into three files in the "nb" directory. The template comes with a Dockerfile so that the resulting application can be easily deployed with Voila.

This template is best used in development environments that support interactive development. The VSCode Python extension has many features that improve the notebooks application development experience. It supports opening and editing Jupyter Notebooks natively. Additionally, it provides a way to view changes caused by updating the source code - you can run Jupyter-like code cells in an environment in tandem to the hard-coded file. This is useful because it allows developers to explore small code changes that affect visual components without having to restart and rerun the application notebook to reflect the changes. Because of this VSCode is an ideal IDE for this kind interactive development taking place within Python files.

The second template was a consequence of an updated design - discussed in the Modularity section - that accommodated increased complexity though a Model-Delegate design pattern. The source code from this template is compiled from a series of Jupyter Notebooks that contain both source code and other cells that aren't included in the final source such as Markdown, tests, and other code that accompany literate programming projects.

This template complements the JupyterLab environment because all development takes place in Jupyter Notebooks. As JupyterLab is an open-source child of Project Jupyter, there are many extensions available to improve the notebook development environment. Some of these replicate common features such as variable expansion and debugging. Not all of these extensions ship with JupyterLab, so we have also developed a Docker image with these extensions already downloaded and activated. The nbdev_app_template [33] was made to utilize this development environment, though it could also be used in an IDE such as VSCode.

V. Deployment Options

Due to its growing popularity, a variety of deployment solutions are now available for Jupyter notebooks. At the simplest level, a set of Jupyter notebooks in a GitHub repository can be made "interactive" and "reproducible" via Binder [18]. Binder turns a repository of notebooks with an optional `requirements` file listing the necessary software dependencies into a web-accessible Jupyter notebook server (running in a Docker container) hosting these notebooks. While Binder provides a public service, it can also be set up separately and used to provide a private service available to authorized users. A variety of gateway frameworks also support JupyterHub or JupyterLab as "first-class citizens". For instance, both the HUBzero [26] and National Data Service Labs Workbench [34] gateway frameworks provide notebook server environments for users to run Jupyter notebooks with custom kernels. Similarly, modern web-based gateways to HPC systems such as Open OnDemand [35] also provide JupyterHub and JupyterLab services that can be used to run Jupyter notebook servers as Slurm jobs on HPC nodes (both CPU and GPU). The rise in commercial and academic cloud provides yet another scalable and extensible deployment option for Jupyter environments. The JupyterHub project [36] now provides heavily customizable Helm charts that can be used to deploy JupyterHub to a Kubernetes cluster on various cloud environments. The provided configuration options for instance allow for customization of the notebook server images available to users, the spawner used to launch these notebook server containers, persistent storage, as well as the integration of CILogon based user authentication and authorization. With this wide variety of deployment options available, the RSE would need to consider both the ease of deployment, as well as the anticipated usage models before settling on a particular solution.

VI. Conclusion

To review, web applications are becoming increasingly popular research communication tools. We have found that Jupyter Notebooks make successful deployment tools because they do not require web development expertise. This makes them easier for research groups to maintain them in the future. Notebook applications have a large variety of data displays and visualization libraries that can often be leveraged into the ipywidget ecosystem, making Jupyter a flexible choice despite the lack of stylistic customization available for widgets. We have developed different design strategies to accommodate differing levels of complexity. For simpler applications, we have provided a template that utilizes a traditional MVC design. For applications benefiting from modularity and composability, we have a template that employs a Model-Delegate design. We have provided guidance for how to approach dynamically-sized arrays. The largest and least-obvious drawback to Jupyter apps is that common third-party data visualization libraries

lack ways to reflect changes from a data visualization back to the model that represents it. bqplot, a maintained member of the ipywidget ecosystem, a provides a solution for a parochial set of interactions. On the other hand, ipyleaflet and other specialized libraries may more successfully provide coverage of bidirectional communication between the view and the model. We have been successful using Jupyter Notebooks in a variety of deployment scenarios, including composable platforms and science gateways.

Jupyter Notebook is not a panacea for scientific web application development, though it can serve as an appropriate stepping stone even for apps that may later evolve to outgrow notebook limitations. Jupyter can be deployed in many ways, making it possible to start out with a simple deployment on Binder, and then one day graduating as a tool on a gateway. It may serve as a easy starting point for iterative development, and then when a limitation has been reached, it is possible to wrap Models with a more sophisticated Python GUI library.

Parameters not discussed here that might make good candidates for further discussion are scalability, user groups, and access to HPC resources. We would also like to extend this dialogue to include concerns such as refactoring when the application has outgrown the limitations of Jupyter Notebooks. As the need for web applications grow, we anticipate encountering new use-cases that will expand the breadth of our knowledge and grow into a more comprehensive set of recommendations.

REFERENCES

[1] J. L. Burnett, R. Dale, C.-Y. Hou, G. Palomo-Munoz, K. S. Whitney, S. Aulenbach, R. S. Bristol, D. Valle, and T. P. Wellman, "Ten simple rules for creating a scientific web application," *PLOS Computational Biology*, vol. 17, no. 12, p. e1009574, Dec. 2021. [Online]. Available: https://dx.plos.org/10.1371/journal.pcbi.1009574

[2] T. Kluyver, B. Ragan-Kelley, P&#233, F. Rez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, n, S. Abdalla, C. Willing, and J. D. Team, "Jupyter Notebooks – a publishing format for reproducible computational workflows," *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pp. 87–90, 2016, publisher: IOS Press. [Online]. Available: https://ebooks.iospress.nl/doi/10.3233/978-1-61499-649-1-87

[3] "Jupyter kernels," Project Jupyter, 2022. [Online]. Available: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels

[4] "xwidgets," Project Jupyter, 2022. [Online]. Available: https://github.com/jupyter-xeus/xwidgets

[5] "ipywidgets," Project Jupyter, 2022. [Online]. Available: https://github.com/jupyter-widgets/ipywidgets/

[6] S. Gowda, P. Vertechi, and J. Mason, "interact.jl," 2022. [Online]. Available: https://github.com/JuliaGizmos/Interact.jl

[7] "ipyleaflet," Project Jupyter, 2022. [Online]. Available: https://github.com/jupyter-widgets/ipyleaflet

[8] V. Agafonkin, "Leaflet," 2022. [Online]. Available: https://github.com/Leaflet/Leaflet

[9] S. Corlay, "bqplot," 2022. [Online]. Available: https://github.com/bqplot/bqplot

[10] M. Breddels, "ipywebrtc," 2022. [Online]. Available: https://github.com/maartenbreddels/ipywebrtc

[11] "xleaflet," QuantStack, 2022. [Online]. Available: https://github.com/jupyter-xeus/xleaflet

[12] "xplot," QuantStack, 2022. [Online]. Available: https://github.com/QuantStack/xplot

[13] "xwebrtc," QuantStack, 2022. [Online]. Available: https://github.com/maartenbreddels/ipywebrtc

[14] A. Rule, A. Tabard, and J. D. Hollan, "Exploration and Explanation in Computational Notebooks," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, ser. CHI '18. New York, NY, USA: Association for Computing Machinery, Apr. 2018. [Online]. Available: https://doi.org/10.1145/3173574.3173606

[15] "Altair," 2022. [Online]. Available: https://github.com/altair-viz/altair

[16] "Bokeh," 2022. [Online]. Available: https://github.com/bokeh/bokeh

[17] "Voilà Gallery." [Online]. Available: https://voila-gallery.org/

[18] P. Jupyter, M. Bussonnier, J. Forde, J. Freeman, B. Granger, T. Head, C. Holdgraf, K. Kelley, G. Nalvarte, A. Osheroff, M. Pacer, Y. Panda, F. Perez, B. Ragan-Kelley, and C. Willing, "Binder 2.0 - Reproducible, interactive, sharable environments for science at scale," Austin, Texas, 2018, pp. 113–120. [Online]. Available: https://conference.scipy.org/proceedings/scipy2018/project_jupyter.html

[19] "Voila gallery," Project Jupyter, 2022. [Online]. Available: https://voila-gallery.org/

[20] W. McKinney *et al.*, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*, vol. 445. Austin, TX, 2010, pp. 51–56.

[21] "Binder," Project Jupyter, 2022. [Online]. Available: https://mybinder.org/

[22] R. Campbell, R. Kalyanam, C. Song, and L. Zhao, "A Template for Rapid Development of Interactive Computing Tools," Oct. 2021, publisher: Zenodo. [Online]. Available: https://zenodo.org/record/5570549

[23] "ipympl," The Matplotlib Development Team, 2022. [Online]. Available: https://matplotlib.org/ipympl/

[24] "ITaP Research Computing - Compute: Geddes." [Online]. Available: https://www.rcac.purdue.edu/compute/geddes

[25] R. Kalyanam, L. Zhao, X. C. Song, V. Merwade, J. Jin, U. Baldos, and J. Smith, "GeoEDF: An Extensible Geospatial Data Framework for FAIR Science," in *Practice and Experience in Advanced Research Computing*. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 207–214. [Online]. Available: http://doi.org/10.1145/3311790.3396631

[26] M. McLennan and R. Kennell, "HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering," *Computing in Science Engineering*, vol. 12, no. 2, pp. 48–53, Mar. 2010, conference Name: Computing in Science Engineering.

[27] R. Eckstein, M. Loy, and D. Wood, *Java Swing*, 1st ed., ser. Java series. Sebastopol, Calif: O'Reilly, 1998.

[28] "traitlets," IPython, 2022. [Online]. Available: https://github.com/ipython/traitlets

[29] "NotebookJS." [Online]. Available: https://notebookjs.app/

[30] "traits," Enthought, 2022. [Online]. Available: https://github.com/enthought/traits

[31] *Leveraging Traits for Highly Interactive Computational Tools in Jupyter*. Zenodo, Oct. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5570605

[32] "functools — Higher-order functions and operations on callable objects — Python 3.10.4 documentation." [Online]. Available: https://docs.python.org/3/library/functools.html

[33] N. Brewer, R. Kalyanam, R. Campbell, C. X. Song, and Z. Lan, "Scientific web applicaiton template," 2022. [Online]. Available: https://github.com/nicole-brewer/nbdev_app_template

[34] C. Willis, M. Lambert, K. McHenry, and C. Kirkpatrick, "Container-based Analysis Environments for Low-Barrier Access to Research Data," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*. New Orleans LA USA: ACM, Jul. 2017, pp. 1–4. [Online]. Available: https://dl.acm.org/doi/10.1145/3093338.3104164

[35] D. Hudak, D. Johnson, A. Chalker, J. Nicklas, E. Franz, T. Dockendorf, and B. McMichael, "Open OnDemand: A web-based client portal for HPC centers," *Journal of Open Source Software*, vol. 3, no. 25, p. 622, May 2018. [Online]. Available: https://joss.theoj.org/papers/10.21105/joss.00622

[36] "Zero to jupyterhub," Project Jupyter, 2022. [Online]. Available: https://zero-to-jupyterhub.readthedocs.io/en/latest/