JupyterLab Integration for NIST Testbed

Yunlan Li*, Artiom Baloian*, Jan Janak*, and Henning Schulzrinne*

*Department of Computer Science, Columbia University, USA
Email: yl4387@columbia.edu, ab4659@columbia.edu, janakj@columbia.edu, hgs@cs.columbia.edu

***Abstract*—The mission-critical voice (MCV) testbed is a hardware and software architecture for conducting human subject experiments with mission critical voice communications. The data collected during experiments is stored on a remote server, but the testbed lacks an environment for researchers to perform statistical analysis on the experiment data. We present a solution using JupyterHub to allow users to run JupyterLab to perform analysis of experiment data and audio files on the remote server directly from their web browser. We configured our JupyterHub deployment with DockerSpawner and a customized HTTP header based authenticator for a seamless integration into the Google OAuth2 authentication system of the MCV testbed.**

I. Introduction

In the age of big data, many data is stored on remote servers or in the cloud. In some situations, the data stored is subject to restrictions on its transfer and use. In these scenarios, one may wish to expose an interface for users to perform analysis of the data on a remote server or in the cloud remotely.

The National Institute of Standards and Technology (NIST) testbed (Fig. 1) is a hardware and software architecture for conducting human subject experiments with mission-critical voice (MCV) communications. It consists of an Intel Next Unit of Computing (NUC) computer and a collection of Raspberry Pi based user terminals with push-to-talk (PTT) speaker-microphones attached, all connected via an Ethernet network that can emulate a radio frequency (RF) channel environment and its impairments (bandwidth limitation, delay, packet loss, bit errors). It is managed via a web-based user interface based on ReactJS. Through the UI, the experimenter can program (script) a communication-based experiment and conduct the experiment with first responder volunteers in a controlled environment. The data collected during the experiment are stored on a remote server (an Intel computer). Since the experiment data contains sensitive information and is subject to Institutional Review Board (IRB) conditions, we wish to allow researchers to analyze the experiment data stored on the remote server directly from the web-based interface of the testbed.

We present an architecture (Fig. 2) based on JupyterHub [1] to allow users analyze experiment data on the remote server directly from their web browser. We used a customized HTTP header based authenticator and a reverse HTTP proxy to integrate JupyterHub into the authentication system of the NIST testbed. In addition, we use DockerSpawner [2] to launch a dedicated JupyterLab [3] instance within a docker container on the remote server for each testbed user. Finally, we use docker volume and bind mounts to achieve data persistence and data sharing between users.

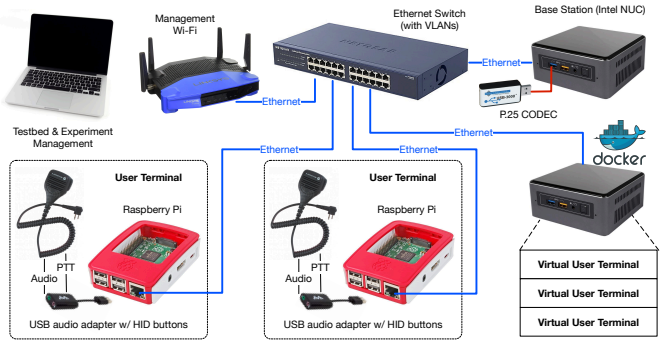

Fig. 1. Testbed architecture

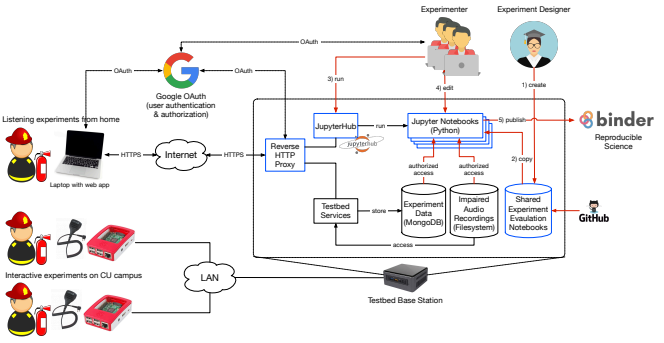

Fig. 2. Architecture of JupyterLab Integration for NIST testbed

The remainder of this paper is structured as follows. In Section III, we present an overview of Project Jupyter [4] and the subsystems of JupyterHub that make managing multiple users' connections to JupyterLab instances possible. In Section IV, we propose an architecture for integrating JupyterHub into a system with an existing authentication system and enabling users to access existing services on the JupyterHub host from within the JupyterLab instance in a docker container.

II. Assignment

JupyterLab is a web-based environment for interactive computing in Python. JupyterLab programs are organized into notebooks. Each notebook consists of a series of steps that can be executed interactively. The ability to generate graphs

directly within the notebook makes JupyterLab popular in the scientific community.

The goal of this project is to find a way to integrate JupyterLab into NIST testbed. Specifically, we would like to:

- Run a JupyterLab instance on the testbed for direct access to sensitive data
- Integrate JupyterLab with the testbed's authentication system (OAuth2)
- Give notebooks access to the data collected during human subject experiments
- Give notebooks access to audio recordings stored on the testbed
- Add an ability to run notebooks on experiments repeatedly

As part of this project, we also want to create a few simple notebooks to demonstrate the integration.

III. Background

Jupyter Notebook is a document specification (.ipynb) file that interweaves code, narrative text, equations and rich output. It also refers to the web-based interface where users can run notebook files [4].

As an extension of Jupyter Notebook, JupyterLab provides a more extensible interface that allows users to run multiple notebooks in separate tabs within a single browser tab.

Both Jupyter Notebook and JupyterLab are intended for single user. Officially, it is discouraged to set up Jupyter Notebook or JupyterLab to allow concurrent access from multiple users because users' commands may collide, clobber and overwrite each other. To serve Jupyter Notebooks for multiple users, the official solution is JupyterHub [5]. JupyterHub provides a framework for providing JupyterLab for multiple users concurrently. It offers many advantages.

Firstly, it provides a solid framework for managing multiple users' connections to JupyterLab concurrently while making it highly customizable. Users can customize key parts of JupyterHub, such as user authentication and spawning of notebook servers, to tailor to their specific needs.

Secondly, since its first preview release in March 2015, it has been actively developed and maintained by a dedicated team. As part of the open-sourced Project Jupyter, it integrates well with JupyterLab and will integrate well with its future releases. This allows us to leverage new features of JupyterLab by simply upgrading JupyterHub, which reduces the risk of having to re-design and implement the architecture for managing multiple JupyterLab instances.

Lastly, JupyterHub could be scaled easily to thousands of users. It has been used in a variety of context, ranging from research labs to university classes.

A. Distributions

Currently, JupyterHub can only be installed on Linux/Unix based systems. As of 2021, there are 2 main distributions of JupyterHub. A JupyterHub distribution is tailored to a specific set of use cases and is easier to set up than setting up JupyterHub from scratch [1].

- Zero to JupyterHub on Kubernetes [6]: for a large number of users, scalable to large number of users and machines
- The Littlest JupyterHub (TLJH) [7]: for 1-100 users on a single machine

B. Subsystems

JupyterHub initiates 3 important types of processes, which are typically referred to as the subsystems of JupyterHub:

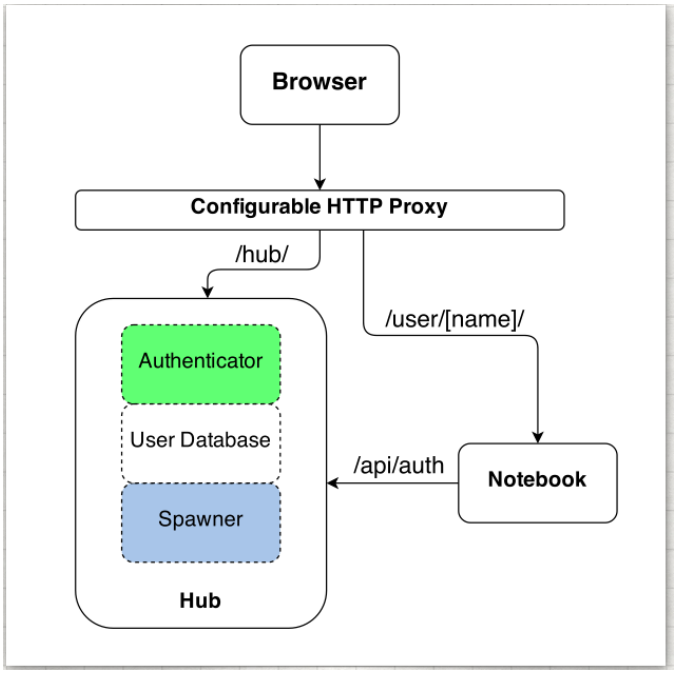

Fig. 3. JupyterHub Subsystems [8]

- **configurable http proxy** (node configurable-http-proxy [9]): proxies all JupyterHub users' requests to other subsystems. This is the public facing part of JupyterHub.
- **hub** (tornado [10]): authenticates hub users and launches Jupyter Notebook server for them. This is the heart of JupyterHub and has many customizable components. Its main job is to control the life-cycle of the notebook server on behalf of the user.
- **single-user notebook server** (Jupyter Notebook): the dedicated per-user Jupyter Server that allows users to create, edit and execute notebooks.

C. Interaction between subsystems

When JupyterHub starts, it launches the configurable-http-proxy and the hub. The single-user notebook server, however, is launched and terminated by the hub on demand for each user. The process through which the hub starts a single-user notebook server is referred to as spawning.

In Fig. 3, `/` represents the base URL of HTTP proxy started by JupyterHub. When a user tries to access any URI that starts with `/hub/`, the proxy will direct the request to the hub, while those starting with `/user/[name]/` will be directed to the given user's single-user notebook server.

Each single-user notebook server communicates with the hub on two important occasions:

- user authentication: the notebook server makes an API request to the hub to identify the user via OAuth the hub authenticates the user by checking the cookie in the request. An authenticated user has the cookie set by the hub on successful login.
- route registration: after successful spawning, the notebook server registers a route at the proxy from which subsequent user requests are re-directed to itself. The notebook server makes an API request to the hub, which then notifies the proxy to add the route.

D. Customization

The flexibility of JupyterHub comes from the possibility and ease of customizing the hub subsystems. JupyterHub encapsulates each key functionality, such as authentication and spawning, in a Python class object and provides a base class from which users can inherit to create their own implementation.

A common paradigm across Project Jupyter is using Traitlets [11] for exposing points of configuration. Traitlets provides a set of configuration classes for programmers to inherit from. The base configuration classes allow users to expose class attributes as configurable options, which are referred to as traits.

Among the advantages that Traitlets offers, the most notable ones are

- **dynamic type checking**: each trait has a type and an error will be thrown if the type of assigned trait value doesn't match. Supported trait types as of Traitlets 5.0 include int, container types such as list and dict, customized classes, etc.
- **ease of configuration**: users can set trait values via command line arguments, JSON file(s) or pure Python configuration file(s).
- **automatic generation of configuration**: Traitlets provide a primitive to easily generate a Python configuration file with all configurable traits of the application, along with help strings. In the case of JupyterHub, a `jupyterhub_config.py` file can be generated under the current working directory simply with

```
$ jupyterhub --generate_config
```

There are a variety of customized components that are available for use directly. In the case of authenticators, JupyterHub comes with PAMAuthenticator where each hub user is also a system user and users are authenticated with their system username and password. An OAuthenticator module is also available to authenticate users via OAuth from OAuth providers such as Google, GitHub, etc. On top of these, there are several authenticators written by the JupyterHub community [12], among the most popular being the jwtauthenticator that authenticates user with a JSON web token.

In the case of spawners, based on whether the notebook server is spawned on a cluster or a single server and if it is containerized, the popular ones are grouped in Fig. 4.

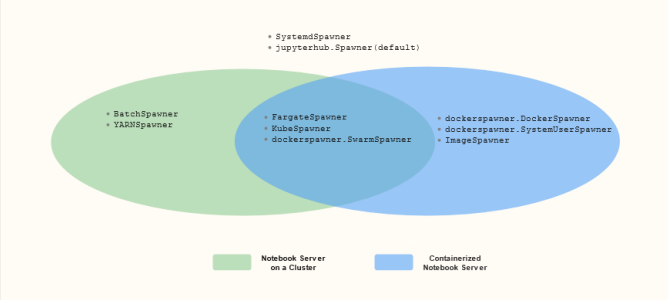

Fig. 4. Grouping of Popular Spawners

The option of spawning notebook servers on a cluster enables JupyterHub to be scaled to support thousands of users. For example, the groundwork of KubeSpawner draws largely from the successful use of JupyterHub and Kubernetes as a hosted computing environment for student assignments at UC Berkeley in their Data 8 Program, which in the recent offering in Spring 2021, has over 1300 students enrolled [13].

IV. System Design

A. Authenticator

The NIST testbed uses Google OAuth2 for user authentication. If we use authenticators such as OAuthenticator and jwtauthenticator, each user needs to be authenticated twice - once for the testbed and another for JupyterHub. This double authentication is redundant and ideally,

- if a user has already authenticated with the testbed, he/she should be able to launch a single-user notebook server directly
- otherwise redirect to the testbed's login page and on authentication success, redirect to the original JupyterHub URL.

To address this issue, we put JupyterHub behind a reverse HTTP proxy, use an authentication middleware and use a customized authenticator that authenticates the user through an HTTP header. The exact mechanism is shown in Fig. 5.

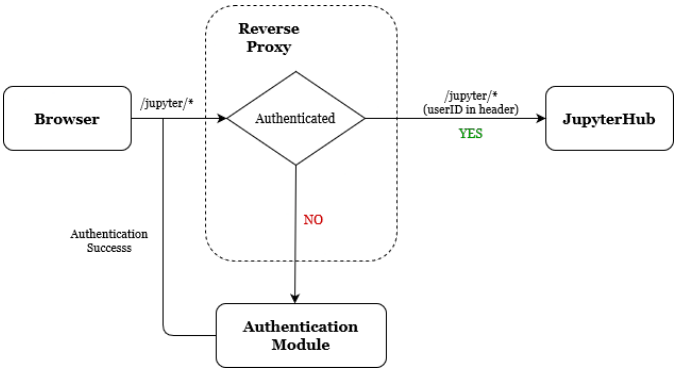

Fig. 5. Integration of JupyterHub into Testbed's Authentication System

Assuming that `/jupyter` is the public facing base URL of JupyterHub. When a user tries to access JupyterHub from the browser, the reverse HTTP proxy uses an authentication middleware that checks if the user has been authenticated with the testbed. If the user is authenticated, it adds to the request an `X-Authorized-User` header with the unique identifier of the user as its value and then forwards the request to JupyterHub. Otherwise, it sends an API request to the authentication endpoint of the testbed with the success URL set to the original request URL.

Under this scheme, each hub user is a verified user on the testbed and the authenticator identifies the user by the `X-Authorized-User` header value, which is then returned by the authenticator and forwarded to the spawner.

Websocket request to the single-user notebook server is initiated when a user tries to open terminals or execute notebooks in JupyterHub. However, it is unnecessary to add an authentication middleware for websocket requests to JupyterHub. If a user hasn't authenticated with the testbed, he/she hasn't been authenticated by JupyterHub, which means that the cookie for JupyterHub is not set in the user's browser. When the user tries to open a JupyterHub URL that will initiate a websocket request, the notebook server's API call to the hub for authenticating the user will fail. Consequently, the user is redirected to the login page of JupyterHub and the reverse HTTP proxy will redirect the user to testbed's login page.

B. Spawner

The NIST testbed uses a single server and each testbed user is not necessarily a system user on the base station. Therefore, DockerSpawner becomes a natural choice, where a notebook instance is launched in a docker container on a single host. In this case, containerization of notebook server is preferred as

- it provides greater isolation from the host, which is beneficial in terms of security as users can execute arbitrary Python scripts in the notebook server.
- it saves the effort to create a system account for each testbed user in advance as docker containers can be created and destroyed on demand.
- creating system account takes up resources and system accounts will not be used for purposes other than spawning notebook servers.

When the notebook server is launched in a container, additional work needs to be done to persist users' data and allow users to access resources on the docker host. This will be discussed in Section IV-C. In addition, it is worth noting that the advantages of containerization of notebook servers comes at the cost of making a per-user Python/Conda environment much more difficult to achieve.

C. Data Access and Storage

Within the notebook server, we wish to give each user access to the MongoDB database and media files on the remote server, persist users' notebooks across container restart, and finally to enable users to easily share notebooks with other hub users.

The services running on the base station is visualized in Fig. 6, with services irrelevant for data access within the notebook servers omitted.

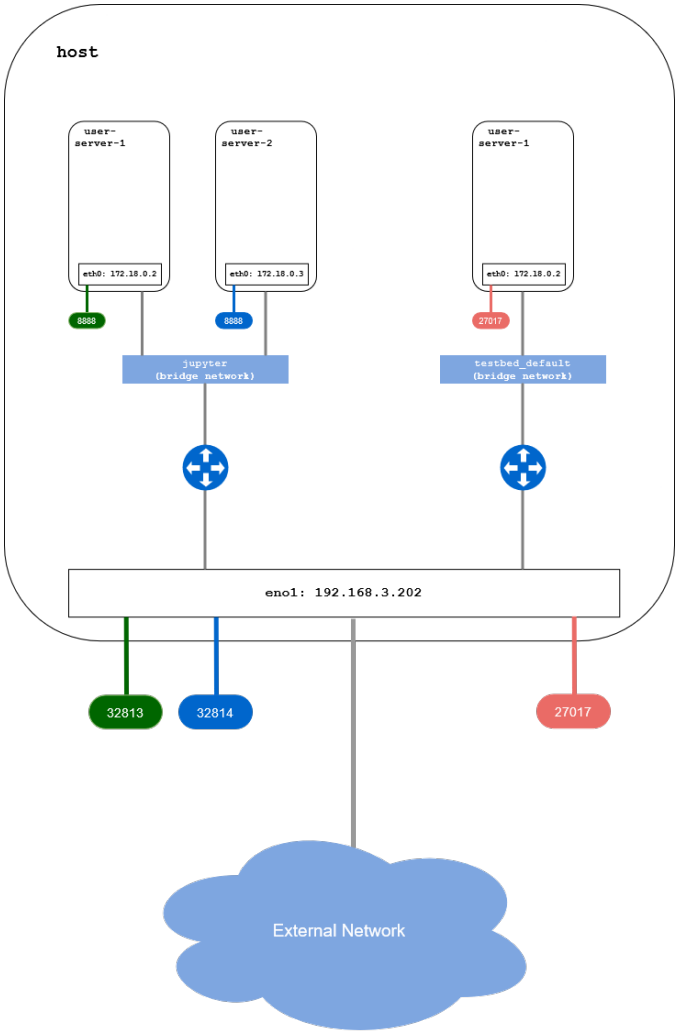

Fig. 6. Services on Base Station

As shown in Fig. 6, all containers running notebook servers are connected to the user-defined docker bridge network `jupyter` where the port on which the notebook server listens on is mapped to a randomly assigned port on the host. The MongoDB database runs inside a docker container on another bridge network `testbed_default` and has the port it runs on mapped to the same port on all network interfaces of the docker host.

To prevent the notebook server from being externally accessible at port `32813` of the host shown in the graph above, we could bind its port to be only accessible from the loopback device `127.0.0.1` by providing the host IP to which it can be accessed in the `-p` flag in the docker start command.

Since MongoDB is accessible on all interfaces of the host on port `27017`, within the notebook server, we can use the pymongo Python package to establish a connection to

host-gateway:27017. Starting from docker 20.10 on Linux, host-gateway is available and resolves to the IP of the network gateway. If an older version of docker is used, we could get around this issue by retrieving the IP in JupyterHub configuration file using the netifaces Python package and pass it along to the container via the extra_hosts key in c.DockerSpawner.extra_host_config in the JupyterHub configuration file.

We use docker bind mount to mount the media file directory on the host to a directory in the container with read only access. This makes the media files available within the notebook server.

Docker volumes are attached to containers and each is mounted into a directory within the container. Files put into the mounted directory are stored on the filesystem of the docker host and is managed by docker. Docker volume is identified by its name. When it is attached to a container, it will be automatically created if it doesn't exist yet. Thus, we could attach a jupyter-username volume, where username resolves to the hub username, to the container running notebook server to persist users' data. Since docker volume can be attached to multiple containers at the same time, we could attach the same volume to each user's container for notebook and file sharing.

```
notebook_dir = '/home/jovyan/nist'
user_file_dir = f'{notebook_dir}/notebook'
media_file_dir = f'{notebook_dir}/media'
share_dir = f'{notebook_dir}/share'

c.DockerSpawner.notebook_dir = notebook_dir
c.DockerSpawner.volumes = {
  'jupyterhub-{username}': user_file_dir,
  'jupyterhub-shared': share_dir,
  '/srv/lmr/wavs': {
      'bind': media_file_dir,
      'mode': 'ro'
  }
}
```

We can specify bind mounts and volumes that need to be attached to the container via the option c.DockerSpawner.volumes. When the configuration file is loaded, {username} will be replaced with hub username. If the key is not a valid path on the filesystem, it is treated as a volume.

To sum up, we use bind mount to allow users to access media files on the remote server within the container. We attach a per-user docker volume to the container for persisting users' data and a shared docker volume for sharing notebooks. Lastly, the MongoDB database is available at host-gateway:27017 within the container.

D. Papermill

When a notebook that analyzes data on one experiment could be applied to a set of other similar experiments, we may wish to run the notebook on another experiment by simply specifying necessary information such as the name of the experiment. In this way, we could automate some data analysis tasks.

We can prepare several base notebooks in advance and use the Python package papermill [14] for parameterizing and executing notebooks. It can be used either through the command line interface (CLI) or the Python API.

An example of papermill usage through CLI:

```
$ papermill local/input.ipynb   \
            local/output.ipynb  \
            -p num 11           \
            -r experiment_run_id f83jh83
```

Local file paths are prefixed with local/. The above command will inject a cell at the top of the notebook named input.ipynb under the current directory with parameters experiment_run_id set to the string f83hj83 and num set to the number 11. It will then execute the notebook and produce an output file named output.ipynb under the current working directory.

V. Conclusion

The scalability and ease of customization and integration of JupyterLab makes JupyterHub an ideal framework for managing concurrent access to multiple JupyterLab instances.

We presented an architecture that leverages JupyterHub to provide concurrent JupyterLab access from web browsers for multiple authenticated users to analyze experiment data and files on a remote server. To integrate JupyterHub into an existing authentication system, we put JupyterHub behind a reverse HTTP proxy and customizes JupyterHub's authenticator to authenticate users via an HTTP header. We used DockerSpawner for spawning notebook servers in a docker container, docker volume to persist users' files in the container and bind mounts to make data on the remote server available inside the container.

References

[1] Project Jupyter team. (2021) JupyterHub - JupyterHub 1.3.0 documentation. [Online]. Available: https://jupyter-server.readthedocs.io/en/latest/

[2] Jupyter Contributors. (2021) DockerSpawner. [Online]. Available: https://jupyterhub-dockerspawner.readthedocs.io/en/latest/index.html

[3] Project Jupyter. (2021) Overview — JupyterLab 3.1.0a10 documentation. [Online]. Available: https://jupyterlab.readthedocs.io/en/latest/getting_started/overview.html

[4] ——. (2021) Project Jupyter. [Online]. Available: https://jupyter.org

[5] Jupyter Team. (2021) Running a public Jupyter Server — Jupyter Server 1.9.0.dev0 documentation. [Online]. Available: https://jupyter-server.readthedocs.io/en/latest/operators/public-server.html

[6] Project Jupyter Contributors. (2021) Zero to JupyterHub with Kubernetes — Zero to JupyterHub with Kubernetes documentation. [Online]. Available: https://zero-to-jupyterhub.readthedocs.io/en/stable/index.html

[7] JupyterHub Team. (2021) The Littlest JupyterHub - The Littlest JupyterHub v0.1 documentation. [Online]. Available: https://tljh.jupyter.org/en/latest/index.html

[8] Project Jupyter team. (2021) Technical Overview — JupyterHub 1.4.1 documentation. [Online]. Available: https://jupyterhub.readthedocs.io/en/stable/reference/technical-overview.html

[9] JupyterHub. (2021) configurable-http-proxy. [Online]. Available: https://github.com/jupyterhub/configurable-http-proxy

[10] Tornado. (2021) Introduction - Tornado 6.1 documentation. [Online]. Available: https://www.tornadoweb.org/en/stable/guide/intro.html

[11] The IPython Development Team. (2021) Traitlets(stable) Documentation. [Online]. Available: https://traitlets.readthedocs.io/en/stable/index.html

[12] JupyterHub Team. (2021) Custom Authenticators for JupyterHub. [Online]. Available: https://github.com/jupyterhub/jupyterhub/wiki/Authenticators
[13] UC Berkeley. (2021) DATA C8 001 - LEC 001. [Online]. Available: https://classes.berkeley.edu/content/2021-spring-data-c8-001-lec-001
[14] papermill. (2021) Papermill GitHub. [Online]. Available: https://github.com/nteract/papermill