Leveraging Notebooks on Testbeds: the Grid'5000 Case

Luke Bertot, Lucas Nussbaum

Leveraging Notebooks on Testbeds: the Grid'5000 Case

Luke Bertot and Lucas Nussbaum
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
firstname.lastname@inria.fr

***Abstract*—Computer science testbeds often require extensive experiment automation to be used efficiently. Jupyter notebooks can contain the scientific reasoning, experiment orchestration, and experiment results in an easy to read, easy to execute format. However some level of support is required from the testbeds to facilitate notebook use on their platforms. Additionally this format can be used for more than driving experiments, and different uses have different requirements in terms of support.**

In this paper we present an analysis of different notebook uses to be expected on computer science testbeds, as found through the feedback of users of the Grid'5000 testbed and a survey of other similar testbeds. Then we present the technical implementation of Jupyter on the Grid'5000 to support those uses.

***Index Terms*—testbeds; computational notebooks; Jupyter; reproducibility;**

I. Introduction

Experiments in the study of distributed systems often require complex instrumentation to automate experiments involving large numbers of machines. On Grid'5000, a computer science testbed based in France, we invite researchers to automate their experiments as much as possible. Grid'5000's usage policy limits use of testbed resources during the work day, in order to ensure availability for small scale interactive experiments and preparatory work, and thus forcing large-scale experiments to be run at nighttime and during weekends.

Tools such as Execo [1] and EnOSlib [2], are available to help users automate their experiments. However automating experiments using a specialized tool is a heavy commitment, and users for the most part appear to automate their experiments using ad hoc scripts. Such scripts, built slowly through trial and error, are often written with no thought towards dissemination or reuse. As such they remain unpublished, and the experiment running process is often only vaguely described in the corresponding article.

Computational notebooks are systems inspired by literate programming, laboratory notebooks, and REPL (Read-Eval-Print-Loop) interactive shells, that mix in a single file prose detailing a scientific process, code implementing that process, and the output of that code. Notebooks are often mentioned as a possible tool against the current reproducibility crisis, for their ability to structure messy scripts into self-contained self-explanatory files. Because notebooks are code-agnostic, the code used in them would often be the same as the one used in ad hoc scripts. This means that the barrier of entry is significantly lower than for other forms of tooling.

The work on this project has been supported by the Horizon 2020 Fed4FIRE+ project, Grant Agreement No. 723638.

Using a notebook does not make experiments reproducible in and of itself, and studies [3] have shown that users must be careful in how they structure and use their notebook for it to be reproducible by other users. However, properly used, they are one of the simplest tools to increase the potential for reproducibility of their experiments. For these reasons we started looking at the possibility of adding Jupyter [4], the most popular notebook tool suite, to the Grid'5000 testbed.

In order to decide what kind of support to add to our testbed we surveyed how other computer science testbeds implemented their own Jupyter support. We also requested feedback from Grid'5000 users concerning their use, real or intended, of Jupyter notebooks on the testbed. The analysis of user feedback and the support for Jupyter in other testbeds is what guided our implementation of Jupyter in order to facilitate as many uses as possible.

In this paper we will present the different uses of notebooks we derived from our surveys, their needs and constraints, and discuss how they apply to testbeds in general and Grid'5000 specifically. We will then explain which uses we aim to support and how we implemented a JupyterHub installation that satisfies those uses, as well as share the technical difficulties we encountered as a result of Jupyter's architecture.

The paper is structured as follows. Section II presents Grid'5000 and the Jupyter stack we aim to implement. Section III details the notebook uses derived from user feedback, and how they apply to testbeds. Section IV discusses the integration of Jupyter in other testbeds. Finally, we discuss technical details of our implementation and the technical difficulties encountered in Section V before concluding.

II. Background

A. *Grid'5000*

The Grid'5000 project [5] was initiated in 2003 by the French HPC (High Performance Computing) research community to provide a large scale highly reconfigurable testbed in order to experiment at scale. Today Grid'5000 has evolved in a testbed similar to Chameleon [6] or CloudLab [7], covering topics like HPC, AI, Clouds, Edge Computing, Big Data. The services it offers are similar to those offered by the aforementioned testbeds [8]: bare metal provisioning, network isolation, monitoring services, etc. It also offers similar hardware: x86 and ARM servers, HPC networks, GPUs, etc.

Grid'5000 infrastructure: The Grid'5000 testbed is distributed over eight sites all over France and in Luxembourg and is maintained by a single technical team. Although user management, access control, and maintenance are performed globally, each site operates with their own clusters (sets of closely interlinked homogeneous nodes), site front-end, and service machines. Sites are interlinked by a dedicated network.

Users for the most part connect via `ssh` to one of Grid'5000 access points, from there they are able to connect to the site front-end. On the site front-end machine users have access to their site-specific homedir and the tools to reserve resources and deploy environments to these resources. Grid'5000's testbed control infrastructure is based on OAR [9] and associated tools. Using OAR, the users can book resources, going from single core to entire clusters, based on relative topologies, called *resource trees*, and filter on the resources' properties.

Alternatively users can interact with the testbed through a REST API. Using the API users can list resources available on every site and interact with the instances of OAR of the different sites. User authentication on the API uses `http-basic-auth` with the user's Grid'5000 credential, except for calls made from one of the site front-ends which are automatically associated to the user using `identd`. This API is often used by tools that automate experiments.

B. Jupyter

During research a lot of code is often generated to collect and process data. Often this code was unpublished, researchers simply described the process in the corresponding article, and even when it was published the code was often relatively undocumented and hard to follow. In proposing their notebook format integrating code and prose, the Jupyter team hoped to foster readable code for reproducibility.

Jupyter notebooks combine ideas of literate programming, interactive programming, and laboratory notebooks. Notebook files contain code cells interspersed with text. Additionally outputs generated by the executed code cells are also saved in the notebook. The jupyter-notebook application is designed to view, edit, and run notebooks and the execution of code cells is handled by Jupyter kernels.

Kernels are language-specific programs similar to interactive shells that execute the code contained in cells and keep program state (variables, functions, ...) in between cells. Because state is only kept in the kernel and not the notebook files, reproducibility of notebook outputs is only guaranteed if cells are executed exactly once and in order.

However if users are careful about properly executing their notebook, and document their process in text cells, notebooks are good supports for reproducible research.

Jupyter tool stack: The Jupyter notebook file format is a json file describing the different cells, their type (code, text, output), and their content, as well as some meta-data such as the kernel language to use, and the cells execution order for the saved outputs. In this article the term *notebooks* will refer to files in this format.

`jupyter-notebook` and `jupyter-lab` are applications used to provide a graphical user interface for notebooks. These application interfaces are accessed through a web browser and provide a file browser, an interface to view, edit, and execute notebooks, and ways to manage currently running kernels. `jupyter-notebook` is the first version of such application and `jupyter-lab` is a more recent successor. In this article we refer to these applications in their different forms as *labs*, so as to avoid confusion with notebooks.

JupyterHub [10] is a web application designed to facilitate the use of notebooks in multi-user environments, hereafter called *hub*. The hub handles user authentication, the starting and stopping of multiple lab instances at the user's request and transparently redirects the user to their lab interface. JupyterHub is structured around modular components that can be adapted to the hub operator's infrastructure.

The authenticator module controls user management and sets the authentication modalities. The default authenticator module uses Unix accounts on the machine, but other modules are available to using ldap or external OAuth providers.

The spawner module controls the lifecycle of lab instances. These lab instances, `jupyter-labhub`, are extensions of the standard lab applications that can only be executed conjointly with a hub. Different spawner modules can instantiate labs in different manners, from starting a new lab on the hub machine, to instantiating new labs in containers on a kubernetes [11] cluster, to using HPC task managers.

Additionally the hub interacts with a reconfigurable http/websocket proxy that is capable of adding and deleting redirections. All incoming connections to the hub or labs are routed through the proxy with routes being added and removed by the hub as needed. For infrastructures where the lab and hub machines might not be routed to the internet, the proxy is the only component users need to be able to connect to.

III. User feedback, notebook uses, and objectives.

Our main objective when we started considering adding Jupyter to Grid'5000 was to offer an alternative for experiment scripts that would foster reproducibility. But before deciding the perimeter of what to implement we wanted to measure user interest and needs. We asked users for their use (existing or intended) of Jupyter Notebooks on Grid'5000. Based on this feedback we established five different possible uses of notebooks on testbeds.

1. Notebooks as experiment drivers. These notebooks run the experiments from beginning to end, starting with resource reservations and going at least to data collection. To support this usage testbeds must provide an environment where resource reservation and interaction with reserved nodes are possible. Additionally some form of storage will likely be necessary to store experiment bulk results, e.g. execution logs and relevant output. To maximize the cross-user reproducibility of such notebooks some attention must be given to the seamless authentication of users. Ideally resource reservations should not require user credentials to be written in the notebook. This can be achieved by preloading credential information in the environment the lab is executed in, or by providing an out-of-band way for the testbed to provide the resource management system with user information. Although this is the use case that had us interested in notebooks in the first place only one user reported interest in this approach during user feedback.

On Grid'5000, site front-ends are the designated environment for such notebooks: they have access to the user site homedir, have connectivity to every node on the platform, and contain the necessary tools to perform resource reservations. Moreover execution on the site front-ends allows for credential-less resource reservation since OAR commands derive user info from the callers Unix account and API calls are automatically matched to users using `identd`. However using site front-ends implies providing users with the ability to choose which site they want to operate on since every site has a separate homedir.

2. Notebooks as experimental payload. The code contained within these notebooks is the core of experiments. These notebooks run on the reserved resources, and either contain or control the computation that is the subject matter of the experiment. They run from the moment they started on the reserved resource until they output the relevant experiment results. To support this usage testbeds must provide a way to execute the notebooks on reserved resources. Testbeds with heterogeneous nodes should provide ways for the users to select on which nodes the notebooks are going to be executed. When virtualization or containerization is used, testbeds need to consider whether they want to provide ways for the user to dimension their resources. Additionally some notebooks might require specific libraries, so testbeds should consider whether they need to provide multiple different environments or ways for users to add to the existing environments. During user feedback, two users indicated using notebooks in this fashion on Grid'5000. We were also made aware that another user had shared a script meant to automate the setup for this usage.

On Grid'5000, satisfying this usage implies providing a way for users to start lab instances inside arbitrary OAR jobs. To guarantee access to any resource the user might need, our installation will have to let the user select any site and resource they want. For environment adaptability we plan on adding Jupyter commands to environments where the user's site homedir is available, letting users customize the environment by adding libraries in their homedir.

3. Notebooks for post-processing. These notebooks are executed after an experiment to process the results. Supporting this usage will be dependent on your testbed's infrastructure and the type of post-processing expected. Some post-processing can be rather computationally intensive or require specific hardware and notebooks implementing these post-processing should be considered the same as experiment notebooks. Others can be executed on more standard hardware. In either case attention should be given to how users store their results, since these notebooks access experimental data. There were no direct testimony of this usage in user feedback.

On Grid'5000, supporting this use case requires being able to execute notebooks on both front-ends and nodes. This is owed to the fact that some forms of storage are only accessible on specific nodes, and to the fact that heavy computations are not authorized on the site front-ends. As such this use case covers the same constraint as the two previous one for us.

4. Notebooks for exploratory programming. Notebooks for exploratory programming are used by users as a form of enhanced interactive shell in order create new code through trial and error. Most of the previously presented usages often start that way and are refined into fully functional notebooks over time. This use case is somewhat transversal to the previous ones as users preparing to run an experiment on a platform need to be able to test their notebooks interactively to smooth out all the kinks. Supporting this kind of usage is done through providing ways to access a lab interface that lets users benefit from the interactive component of notebooks. Although this usage is not always differentiable from the previous ones, two users reported using notebooks specifically to figure out parameters to specific machine learning algorithms before running the model in a non-notebook fashion.

On Grid'5000, it is already possible to use a VPN or ssh tunneling to connect to lab applications running on front-ends or nodes. However some users have expressed difficulties with using those tools, pushing us to explore the possibility of providing native web access for labs.

5. Notebooks as tutorials. Notebooks as tutorials are notebooks provided to the users by teachers that aim to present and explain to the users some specific concept. As with the previous usage these notebooks rely extensively on the interactive component of lab applications. Since students might not be familiar with the ways one interacts with a given testbed, this usage is best served by providing full web access to lab instances through a JupyterHub-like system. Teachers that use Grid'5000 have been using notebooks in their courses, and have expressed interest for a JupyterHub type access that would reduce the need to guide students through the node-reservation process, setting up the VPN, and manually installing and starting a Jupyter lab.

After considering the demands for these different use cases, we decided that Grid'5000 should implement a fully web-based access allowing users to start a lab on any arbitrary node or front-end from a single interface.

IV. Related work

While planning our own implementation of Jupyter we searched for other computer science testbeds that implement Jupyter notebooks as a feature.

Chameleon, a testbed similar to Grid'5000 based on Openstack [12], implemented their own Jupyterhub instance in 2019 [13]. In Chameleon's implementation JupyterHub starts labs in docker containers. Notebooks executed on these labs can perform resource reservations and drive experiments; Chameleon's implementation uses environment variables containing the user's Openstack credentials to facilitate this process. Chameleon also provides a notebook library for users to share experiments in.

The Minnesota Supercomputing Institute (MSI) also implemented a JupyterHub instance [14] that runs lab instances on HPC computing nodes using the Torque batch scheduler.

Although they have not published any article on the subject, Jupyter is also available on the following testbeds:

– As part of the Fed4FIRE European testbed federation, imec's GPULab testbed proposes a docker powered JupyterHub instance (website: [15]). Lab instances are executed in gpu enabled containers, and the hub offers different container images with a variety of software stacks (R, tensorflow, spark, julia). Users can request specific container sizes from the spawner page, and a separate interface can be used to active port-forwarding between the containers and the internet.

– GriCAD, the University of Grenoble computing center, has a JupyterHub instance capable of starting lab instances on a dedicated machine (website: [16]). This testbed also supports running labs on computation nodes, but such instances are only accessible through ssh tunnelling.

– The IDRIS, a national supercomputing center in France with stringent security measures, bypasses the use of JupyterHub by providing a modified version of `jupyter-lab` that adds itself to a reconfigurable proxy at startup providing a way for users to access their lab instance (website: [17]). These lab instances can be executed both on a front-end machine and computing nodes, but require manual reservation of resources using slurm beforehand.

To the best of our knowledge, Grid'5000 is unique in that we are implementing in a single instance of JupyterHub the ability to spawn lab instances both on reserved resources and front-ends machines, when other testbeds will usually do one or the other.

V. Implementing JupyterHub on Grid'5000

A. General setup

We decided to deploy a single JupyterHub instance for the whole of Grid'5000. This instance is placed on a service machine in the internal Grid'5000 network, with a reconfigurable proxy executed on the same machine as the hub. To allow users access to the reconfigurable proxy, and thus the hub and labs, a route is added to Grid'5000's pre-existing Apache proxies. This frontline proxy is already used for other Grid'5000 services, can handle user authentication, and will mitigate any weakness the reconfigurable proxy might have.

The authentication module used in our hub is `jhub_remote_user_authenticator` [18], which removes the authentication page and instead relies on incoming HTTP headers for authentication. This allows our frontline proxy to perform authentication and pass on authentication information to the hub through http headers. This is advantageous as it uses Grid'5000 already established authentication infrastructure without involving a new service.

The spawner module is a custom module implemented specifically to match Grid'5000's needs, called `G5kSpawner`. From the user point of view the spawner requires the user select a site and whether to start the lab on a front-end or a node. If a node is selected, the users can specify the OAR resource tree to request, the walltime of the OAR job and other information required by OAR to service the request.

Spawner modules are required to implement three functions:

- **start** used to start a new instance of the lab application, return the hostname (or ip) and port of the lab to the hub;
- **poll** used to query the status of a given lab instance, returns northing if the lab is still running or the lab's exit code if it is not;
- **stop** used to shutdown an instance of the lab application, returns nothing.

Additionally the spawner can store the information necessary to track lab instances across hub reboots to persistent storage.

G5kSpawner contains two different implementations of each of the three main functions and calls upon one or the other depending on whether the users request a node or a front-end.

B. Execution on nodes

For the execution of lab instances on compute nodes the hub delegates the execution of the `jupyter-labhub` program to OAR, that it controls through Grid'5000's REST API.

The REST API allows the hub to interact with OAR instances of every site and as a service operated by Grid'5000, the hub is provided an SSL client certificate allowing it to make calls to the API in the name of the user requesting the lab instance. When requesting resources from OAR the hub provides the lab command to execute. OAR will execute the command automatically once the requested resources become available on the main node of the request. If the jupyter-labhub command ends before the end of the OAR job, the job will be ended immediately by OAR. As such the lifecycle of the OAR job and the lifecycle of the lab instance are closely interlinked and the hub treats them as one and the same. Under this mode of operation the three main functions operate as follows:

start: The spawner selects a port on which to run the lab, and prepares the lab command. This command along with OAR information such as the requested resource tree and walltime, are POSTed to the `/sites/<SITE>/jobs` endpoint to create a new OAR job, and the spawner gets a job-id from the reply. The execution site and job-id are saved to the spawner state and added to persistent storage. The spawner monitors the status of the job using the `/sites/<SITE>/jobs/<JOB-ID>` endpoint. Once the job is scheduled this endpoint also provides the hostname of the reserved node to the spawner which is returned along with the selected port to the hub.

poll: The spawner checks the status of the OAR job using the `/sites/<SITE>/jobs/<JOB-ID>` endpoint. If the job state is *finished* or *error* the function returns 254 in lieu of an exit code. For any other status value the lab is considered to be still running and the function returns nothing.

stop: The spawner issues a DELETE call to the `/sites/<SITE>/jobs/<JOB-ID>` endpoint, requesting the end of the OAR job.

The close interlink between the lab and the corresponding OAR job, the REST API, and the existence for the python-grid5000 [19] library used to interact with the API greatly simplifies the code used when instantiating labs on nodes.

C. Execution on site front-ends

For the execution of lab instances on a site front-end the hubs need to connect via ssh to the requested front-end and start the new lab process as the requesting user.

Unlike with the REST API, Grid'5000 has no preexisting method of acting on a front-end as an arbitrary user. And since the hub cannot be authorized to connect via ssh as any user, a new dedicated user was added to the front-end machines for the hub to connect to. This hub user is authorised to run a single script, `jupyterctl`, as root. The `jupyterctl` script act as a wrapper around Jupyter commands, performing the necessary user switching while limiting the capabilities of the hub user to the strict necessary. `jupyterctl` uses sudo to run `jupyter-labhub` and `kill` as other users while switching context to their homedirs. Additionally the script will only agree to poll or stop Jupyter processes. In this mode the three main functions operate as follows:

start: The spawner prepares the environment variables and the arguments for the `jupyter-labhub` as usual, but uses `sudo jupyterctl start` as a command instead. Additionally the `port=` argument is removed by the spawner. The command is executed via ssh on the selected site front-end. On the front-end the `jupyterctl` command scans the machine for a free port and adds the corresponding `port=` argument. Then the `jupyterctl` script extracts the target user from the environment variables set by the hub and starts `jupyter-labhub` as the target user in their homedir, making sure to pass along all the environment variables and arguments set by the hub. Finally the `jupyterctl` script outputs the `jupyter-labhub` process-id and selected port. The process-id and the selected site are added to the spawner's persistent state. The spawner returns the front-end hostname and the selected port to the hub.

poll: The spawner connects to the selected site and runs the `sudo jupyterctl poll <process-id>` command. The script outputs the number of Jupyter processes matching the process-id. If none were found the poll function assumes the lab instance is dead and returns 254 in lieu of exit code, otherwise the lab is still running and poll return nothing.

stop: The spawner connects to the selected site and runs the `sudo jupyterctl stop <process-id>` command while passing the target user in an environment variable. The script checks that the target process-id is a Jupyter process, and uses `sudo kill` as the target user. The script returns the number of matching processes after the kill, and the stop function returns nothing.

This code used in instantiating labs on front-ends is more complex and split between the G5kSpawner and the jupyterctl script. Part of these complexities is due to assumptions made by JupyterHub and a dead-lock between the hub and the lab. We will expand on these circumstances in a later section.

D. User experience

From the user point of view accessing the hub is done in the same way as any Grid'5000 interface. The same dialogue is used for authentication as in other services, and users do not need to log in a second time in JupyterHub. The single JupyterHub instance allows the user to interact with every site, and for node deployment the user is free to choose their resources using the same syntax as they would when connected via ssh. After pressing the start button, the user waits from a few seconds to a few minutes before being seamlessly redirected to the Jupyter lab interface.

The lab instance provided through JupyterHub comes with two kernels, one for python3 code and one for bash code. Users in need of different kernels can use `pip` to install them in their homedir. Kernels installed in this manner, as well as any python library the user may need, will become available in the lab environment.

Since labs on the front-ends are instantiated as the correct user, OAR commands and calls to the Grid'5000 API will work seamlessly without requiring any credential information. Because of this, no sensitive information needs to be stored in the notebook and notebooks should work without modification when shared between users.

Labs instantiated on nodes are started on the Grid'5000 standard environment, in which the user's site homedir is automatically loaded via NFS, making any additional kernel and libraries they might have installed available.

The user's resource usage is kept under the same checks they would be for any other usage of Grid'5000. Node reservations for the purpose of running notebooks fall under the same limits as any other node reservation. Front-ends are shared by nature, and limits on resource usage are enforced using *cgroups*. Labs access the user's homedir that are limited in size and only accessible by the users themselves.

E. Difficulties encountered

1) jupyter-notebook tooling and jupyter-lab: The `jupyter-notebook` provides a set of subcommands to manipulate running instances of `jupyter-notebooks`, giving users the ability find all running instances, and stop specific instances from any terminal. If the *list* subcommand includes instances of `jupyter-labhub`, the *stop* subcommand does not appear to work in our setup. This is the reason we decided to use a `kill` command to terminate instances of `jupyter-labhub`.

2) Remote spawners and specification compliance: The structure of JupyterHub spawner modules seems to favor local processes, or remote processes under highly controlled monitoring systems. Most notably the fact that the **poll** function is meant to recover the exit status of the lab process might be trivial for local processes, and possible when using docker or a task manager such as slurm or OAR, but becomes complicated for for processes open on remote machines started as background tasks under `sudo`. Fortunately JupyterHub doesn't seem to use the exit code in any capacity, and is mostly interested in knowing whether the process is alive or dead.

3) Hub-side port selection and port deadlock: Another problem with remote spawning is JupyterHub's tendency to select the port to use. The base implementation of the spawner, on which most other implementations depend will select a free port on the hub machine to add as a `port=` argument when building the lab command. Port selection is important since the **start** command must return the port on which the instance is running along with the hostname of the instance. The obvious problem being that a free port on the hub machine might not be free on the front-end.

As is standard, setting the `port=0` argument on the lab command would result in the lab selecting a random free port on the machine on which it is running. And using the `jupyter-notebook` list subcommand it is possible to retrieve the port of an instance whose process-id is known. However trying to use this approach with `jupyter-labhub` leads to a deadlock. To appear in the list subcommand an instance of `jupyter-labhub` must be fully initialized, and initialization requires the lab instance to establish contact with the hub. Parallelly the hub is still waiting on the port and hostname information from the **start** command, and will not accept connections from unknown labs. Hence we find ourselves in a situation where the hub is waiting on the spawner for the port number, the spawner is waiting for the lab to appear in the list, and the lab cannot appear in the list until it has established contact with the hub.

To avoid the deadlock while limiting port conflict, the port selection for front-end instances is performed by the `jupyterctl` script, using `shuf` to generate a list of randomized ports and `netcat` to scan if the ports are available.

VI. Conclusions

In this article we presented the integration of Jupyter notebooks to the Grid'5000 platform. We showed our analysis of the different uses notebooks could have on an experimental computing testbed, how other similar testbeds covered these use cases, and presented how our JupyterHub integration covers these uses.

Discussion with Grid'5000 users was enlightening concerning the scope of use notebooks already had on the testbed. We would recommend any testbeds looking to support notebook to survey their users to understand the important uses to cover.

Despite some architectural oddities, JupyterHub has shown itself to be extremely adaptable. The wealth of modules available cover most standard installation needs, and custom modules are relatively easy to build even for uncommon infrastructures like ours.

It is too early to say if notebooks will significantly impact reproducibility of experiments in computer science. Notebooks are not inherently reproducible, and good practices, such as those described in [3], are necessary to guarantee reproducibility. But the adoption of such good practices can only happen if platforms first lower the entry barrier to notebooks to that of ad hoc scripts.

References

[1] M. Imbert *et al.*, "Using the execo toolkit to perform automatic and reproducible cloud experiments," in *CloudCom*, 2013.

[2] R. Cherrueau *et al.*, "Enosstack: A lamp-like stack for the experimenter," in *CNERT*, 2018.

[3] J. F. Pimentel *et al.*, "A large-scale study about quality and reproducibility of jupyter notebooks," in *MSR*, 2019.

[4] T. Kluyver *et al.*, "Jupyter notebooks-a publishing format for reproducible computational workflows.," in *ELPUB*, 2016.

[5] D. Balouek *et al.*, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, 2013.

[6] K. Keahey *et al.*, "Chameleon: A scalable production testbed for computer science research," in *Contemporary High Performance Computing*, CRC Press, 2019.

[7] R. Ricci *et al.*, "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications," *;login:*, vol. 39, no. 6, 2014.

[8] L. Nussbaum, "Testbeds Support for Reproducible Research," in *ACM SIGCOMM 2017 Reproducibility Workshop*, 2017.

[9] N. Capit *et al.*, "A batch scheduler with high level components," in *CCGrid*, 2005.

[10] Project Jupyter team. (2016). Jupyterhub 1.3.0 documentation, [Online]. Available: https://jupyterhub.readthedocs.io/en/stable/.

[11] A. Verma *et al.*, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.

[12] O. Sefraoui *et al.*, "Openstack: Toward an open-source solution for cloud computing," *International Journal of Computer Applications*, 2012.

[13] J. Anderson and K. Keahey, "A case for integrating experimental containers with notebooks," in *CloudCom*, 2019.

[14] M. Milligan, "Interactive HPC gateways with jupyter and jupyterhub," in *PEARC*, 2017.

[15] imec iLab.t team. (2020). Jupyterhub at imec ilab.t - imec ilab.t documentation, [Online]. Available: https://doc.ilabt.imec.be/ilabt/jupyter/.

[16] GriCAD. (2020). Les notebooks jupyter : User documentation for gricad services. French, [Online]. Available: https://gricad-doc.univ-grenoble-alpes.fr/en/notebook/.

[17] IDRIS. (2020). Idris - jean zay: Access to jupyter notebook and jupyterlab with tensorboard, [Online]. Available: http://www.idris.fr/eng/jean-zay/pre-post/jean-zay-jupyter-notebook-eng.html.

[18] C. Waldbieser *et al.* (2019). Jupyterhub remote_user authenticator, [Online]. Available: https://github.com/cwaldbieser/jhub_remote_user_authenticator.

[19] M. Simonin. (2019). Python-grid5000, [Online]. Available: https://gitlab.inria.fr/msimonin/python-grid5000.