A Case for Integrating Experimental Containers with Notebooks

Jason Anderson*† and Kate Keahey†*
*Consortium for Advanced Science and Engineering
University of Chicago, Chicago, IL 60637
Email: jasonanderson@uchicago.edu
†Argonne National Laboratory
Lemont, IL 60439
Email: keahey@anl.gov

***Abstract*—Computational notebooks have gained much popularity as a way of documenting research processes; they allow users to express research narrative by integrating ideas expressed as text, process expressed as code, and results in one executable document. However, the environments in which the code can run are currently limited, often containing only a fraction of the resources of one node, posing a barrier to many computations. In this paper, we make the case that integrating complex experimental environments, such as virtual clusters or complex networking environments that can be provisioned via infrastructure clouds, into computational notebooks will significantly broaden their reach and at the same time help realize the potential of clouds as a platform for repeatable research. To support our argument, we describe the integration of Jupyter notebooks into the Chameleon cloud testbed, which allows the user to define complex experimental environments and then assign processes to elements of this environment similarly to the way a laptop user may switch between different desktops. We evaluate our approach on an actual experiment from both the development and replication perspective.**

1. Introduction

Scientific computations, whether large-scale simulations, just-in-time analytics, or performance studies, each represent a computational *experiment*, which may have to be repeated or replicated [1] for purposes of validation, variation, or other forms of evaluating and leveraging scientific contribution. These computational experiments take place in *experimental containers* that define the environment of the computation in terms of its configuration and makeup (i.e., compute nodes, networks, or storage resources as well as software configuration required by the computation). Such experimental containers may be complex, and thus difficult to create. Changes in their configuration may introduce inconsistencies, lead to wrong results, or even make repeating a computation impossible. Repeating or replicating computational experiments thus relies on ensuring that its experimental container is configured correctly every time.

The task of documenting an experimental process in such a way that it can be shared and repeated has been greatly facilitated by the advent of computational notebooks, which have already proven themselves instrumental in communicating scientific discoveries [2]. Their success lies in the ability to provide a link between *ideas* (explained in text), *process* (captured as code), and *results* in the form of graphs, pictures, or other artifacts. They have however a significant limitation in that the execution environment of a notebook is constrained to a small set of possibilities, such as preconfigured virtual machines or software containers, and often only granted a small fraction of a system's full resources. While sufficient for many generic computations, it falls short of being able to support large-scale scientific experiments that increasingly require complex experimental containers composed of diverse and powerful resources distributed over complex topologies.

The challenge of supporting such environments has been addressed by infrastructure clouds, from commercial resources such as Amazon Web Services (AWS) [3] to open testbeds such as Chameleon [4]: they allow users to create experimental containers implemented using isolation vehicles from virtual machines to bare metal and from nodes to networks. Furthermore, the digital artifacts that represent such containers—such as images and orchestration templates [5] that marshal those images into complex environments such as virtual clusters—can be shared and used by others to reestablish the experimental environment within a given platform. This makes the task of replicating computational experiments significantly easier; but the process that creates those digital artifacts is still complex and thus poses a barrier to publishing them in the first place.

In this paper, we propose a way to integrate notebooks–a highly effective tool for the description of experimental process–and experimental containers, capable of providing complex experimental environments, to mutual benefit. On one hand, dynamically and accurately deployed experimental containers based on shared artifacts developed for clouds or testbeds can address a deficiency of notebooks in terms of documenting the experimental process. On the other hand, notebooks can provide a convenient and intuitive tool for utilizing such containers in a way that more closely resembles the creative process and provides a way to introduce variation in both the container makeup and the steps of

experimentation. To support our argument, we describe the integration of Jupyter [6] notebooks with the Chameleon testbed and a design pattern that leverages this integration to allow the user to define arbitrarily complex experimental containers and then assign processes to elements of this container similarly to the way a laptop user may switch between different desktops. We evaluate our approach on an actual experiment from both the development and replication perspective to illustrate how it can help more investigators structure more complex experiments interactively, and allow for their replication and controlled variation.

This paper is organized as follows. In Section 3 we define *experimental containers*, describe their properties, and discuss their implementation across a range of testbeds. In Sections 4 and 5 we explain how we integrated this concept of a container into notebooks and discuss our implementation of integration of Jupyter notebooks with the Chameleon testbed. In Section 6, we validate our proposal in the context of a demanding distributed experiment and explain its benefits and limitations.

2. Related Work

The value of interweaving ideas (text) and process (code) in one document was first explored in an approach called *literate programming* [7]. Interactive notebooks follow the same pattern, but additionally allow the user to pause, rewind, replay and resume a computation at any point, supporting a more organic workflow. Additionally, the user can work with rich, potentially interactive visualizations. The two predominant implementations of these interactive notebooks are Wolfram Mathematica [8] and IPython (later named Project Jupyter) [9]. Despite Mathematica's longer existence, Jupyter has better penetration within the academic community, due to leveraging a large open-source development community and providing a web-based interface requiring no specialized client tools. The web interface spawned several managed "Notebook-as-a-Service" platforms to aid in reproducibility, such as CodeOcean [10], WholeTale [11], Nextjournal [12], Binder [13], Wolfram Cloud [14], Azure Notebooks [15], Amazon EMR [16], and Google Colab [17]. Pangeo [18] is recent development that applies this pattern to science problems across various domains. Many researchers have experimented with using Jupyter notebooks specifically to improve reproducibility [19], [20], [21], [22], [23].

While compelling in their integrative approach, existing notebook implementations are difficult to use with advanced configurable hardware environments. On managed platforms, the execution environment of a notebook is limited to a small set of possibilities, such as a virtual machine [24] or Docker [25] container. Typically, a specification of the hardware provided is not visible to the user. This approach trades flexibility for simplicity, and is well-suited to research where specifics of hardware configuration are less important e.g., a data processing workflow or simulation. To get around this limitation, a researcher may need to provision their own interactive server that places notebooks on hardware they control. This appraoch requires additional operational knowledge and reduces the likelihood of replicating the experiment results by requiring that future researchers possess similar hardware.

A different approach is implemented by systems such as Popper [26] and Sumatra [27] which express an experimental workflow that can be implemented across platforms in a declarative form. However, the "process" of experimentation, when defined declaratively, requires understanding the syntax and rules of different platforms and also creates challenges for an iterative workflow: declarative systems usually do not have a mechanism for storing intermediate state, and so are biased towards re-running the entire workflow from start to finish. Besides this limitation, researchers find complex scenarios hard to represent unless care and time is taken to follow the declarative model from the beginning [28], [29]. If a researcher makes the investment into such a workflow, their exact process does indeed become reliable, streamlined, and easy to repeat. However, it is still hard to interact with and produce variation easily which is necessary for exploring different inputs, configurations, and hypotheses on the fly; in this context an imperative approach usually works better.

3. Experimental Containers

Most computation takes place in the context of a complex configuration of resources including not only compute nodes, but potentially also accelerators, networks, and storage. Recording and then replicating this configuration is often not only complex and thus hard to do, but also needs to be done accurately to ensure consistency of results. The latter is particularly true of Computer Science experiments in fields such power management or performance variability, but can also affect other types of computations. While virtualization techniques, such as virtual machines or networks, addressed this problem partially, we often need a generalized "container" that extends the abstraction over a combination of multiple resources working in concert. We describe below the properties of such *experimental containers* and discuss several implementations that satisfy them to various degrees.

Isolation: The ability to faithfully replicate a result in a shared environment often relies on the ability to control interference from other users of the system, whether in terms of configuration requirements or computational noise they generate. In [30] we introduce the terms *system isolation*, which presents to users an independent system (as implemented in e.g., virtual machines (VMs) [24] or GENI slices [31]) and allows them to configure an environment unique to a computation, and *performance isolation*, which ensures that the experimental container presents consistent performance (as implemented via e.g., allocating bare metal nodes). What specific type of isolation is required depends on the nature of experiments run in a given container.

Expressiveness: It should be possible to describe an experiment container such that it covers the broadest possible range of different experimental configurations (sometimes called "topologies"), that are both general (e.g., cover complex experimental configurations expressed in a flexible manner), and precise (i.e., describe the required resources with enough detail). This implies the ability to *map* a container onto sets of platform resources according to requirements ranging from very broad (e.g., "at least 2GBs of memory per allocated core"), to more specific (e.g., "two nodes on the same rack"), to unique (e.g., "*this* specific node"). Furthermore, the container description should allow users to assemble nodes, networks, and other units applying the broad and precise principle to all of them.

Integration: A complex experiment container frequently needs to combine resources of different types such that they work as a coherent unit. For example, individual compute resources may need to be combined into a cluster which involves orchestrating a secure exchange of information assigned at deployment time such as IP addresses or security keys; this is accomplished by a technique called *contextualization* [32] or by finalizing configuration when this information becomes available. Multiple virtual networks and wide-area circuits may need to be combined into one virtual network; this is accomplished via *stitching* [31]. Finally, resources of different types need to be configured such that they work with each other in ways that support the needs of an experiment.

Persistence: Once assembled and integrated, it is important that an experimental container can be made persistent for at least some amount of time allowing experimenters to save their work such that the container can be redeployed. There are two factors that affect the ability to persist a container: *fundamental/technical* factors, i.e., the ability to accurately represent the environment such that it can be reestablished (e.g., snapshotting), and *policy* factors, which guide how long the hardware, firmware and software needed for the environment to be reestablished may be available. While the former is a property of a platform, the latter is an economic decision guided by cost considerations (e.g., backwards compatibility of testbed services) on one hand, and specific needs on the other (e.g., replicating an experiment exactly may be most valuable for a period of review only). In fact, as time passes it is often the case that the interest in reproducibility is greater than replicability/repeatability [33], [34].

Shareability: To replicate, multiple users and user groups must have access to resources compatible with the experimental artifacts that the container relies on, such as appliances/images or orchestration templates. This allows reviewers to easily reestablish experiments in papers submitted by authors and enables researchers to compare work done by their colleagues. Open access is thus a critical requirement. While the reach of experiments (via shareability of the containers) could be extended by compatibility of various platforms, there is an inherent tradeoff between generalization and innovation: diverse individual platforms encourage rapid development of new features as science-driven experimental needs evolve; strict portability requirements slow this development.

3.1. Experimental Containers in Chameleon

The Chameleon open testbed serves the needs of Computer Science community; thus its container implementation reflects this community's needs. Performance and system *isolation* needed to support studies of e.g., performance variability or power management strategies are supported by allowing users to provision and reconfigure bare metal servers; users can also provision configurable layer-2 networks for exclusive use. Chameleon's (Blazar [35]) service, used for resource allocation, supports *expressive* constraints-based descriptions allowing users to allocate resources from nodes to networks based on both broad (e.g., collection of nodes within the same physical rack), and detailed constraints (e.g., specific node) descriptions. Allocating collections of resources and *integrating* them (e.g., provisioning a network topology and server configurations over multiple sites, and contextualizing them) is handled by the (Heat [36]) orchestration service which takes an orchestration template specifying the desired configurations as input. The *persistence* aspect is addressed by support for snapshotting disk images, and testbed versioning that tracks (slow) hardware and firmware changes so that users can integrate them into their experimental strategy. Finally, the testbed is open to all CS researchers, so that these persisted artifacts can be *shared* with wider community, and supports such sharing via mechanisms such as e.g., a catalog of images and orchestration templates.

3.2. Other Implementations

Different platforms support different types of experimental containers with varying levels of support for the properties outlined above, as necessary for the needs of their respective communities.

Open Computer Science testbeds often specialize in the support for a specific type of research such as e.g., GENI [31] and PlanetLab [37] for networking, COSMOS [38] and POWDER [39] for wireless, and Grid5000 [40], and CloudLab [41] for cloud computing and datacenter management and support experimental containers that are relevant to the needs of their respective communities. Of those Grid5000 and CloudLab are most similar to Chameleon: both support bare metal provisioning of resources, i.e., performance isolation for compute nodes and Chameleon and CloudLab support the GENI concept of *slicing* [31] to create virtual networks. The testbeds are typically accessible to the broad scientific community making sharing of artifacts within that community possible.

Commercial clouds such as Amazon Web Services and Google Cloud Platform provide a viable alternative for experimental containers. However, while bare metal provisioning is in ascendance notably in AWS [42], on the whole these clouds offer weaker isolation as they predominantly rely on virtualization. In addition, thought they do support building blocks for network isolation such as DirectConnect [43] and ExpressConnect [44], in practice those are difficult to access for individual users. Limited control over hardware mapping makes some experiments difficult or impossible to express, and no insight into hardware versioning makes managing persistence difficult. While their shareability is in principle higher than open testbeds, in practice this is often unrealized due to high monetary cost [45].

3.3. Discussion: Building Experimental Containers

Creating an experimental environment is one thing; doing it easily and in a way that supports the creative process of both the original experimenter and the replicating experimenter is another.

A common solution to the problem of creating a complex experimental environment is a mechanism that allows a user to define the desired end-state of an experimental container declaratively via a template using a special grammar, such as XML used by RSpec [31] on GENI and CloudLab, YAML used by Heat [36] on Chameleon and other OpenStack clouds, or AWS CloudFormation [46] in the realm of commercial clouds. This is often referred to as *orchestration* and allows users to express the shape of an experimental container in terms of resources (i.e., nodes, networks, etc.) that they will use, provides mechanisms for integration, as well as a way to start an execution on boot, theoretically making the execution of the entire experiment an extension of container deployment.

However, while orchestration is currently the most popular solution for building complex experimental containers, it ultimately fails in the support for creative process critical to reproducibility. For one, orchestration is not interactive: it is effectively a transaction, which either commits, obtaining the desired state, or rolls back. Users have limited ability to influence the intermediate state, adjust variables, or rearrange the topology, and are forced into workflows that require re-running the entire transaction, which can take significant time. This illuminates another problem: the orchestration syntax is typically *declarative*, while the creative process is expressed *imperatively* ("first, I did this, then I did this."). Finally, the existing orchestration tools and disk images have limited portability so that while it is advantageous to be able to use the same tool to define both the experimental container and the experimental process, there are also advantages to being able to separate them when needed.

We take two lessons from all of this. First, a more human-centric approach is desired: while an experimental container is inherently a product, creating it is still a process. Thus, it should be possible to both reestablish the experimental container as a transaction and introspect, and possibly vary the process. Second, a higher-level abstraction of the experiment being performed is necessary, one that enables the *execution* of the experiment to be reproduced without modification, even if the *instantiation* of the experiment differs across platforms.

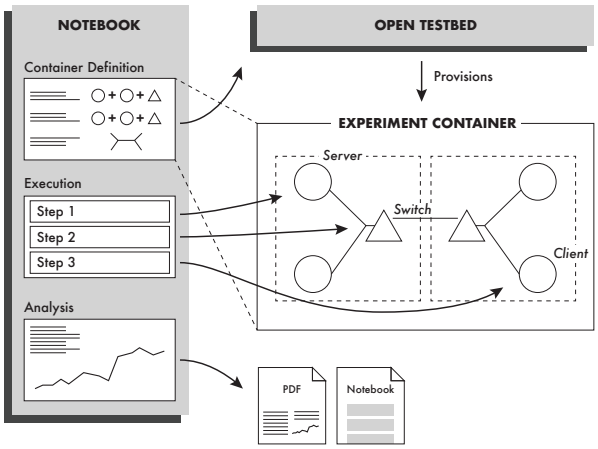

Figure 1. An illustration of our approach: the notebook contains sections setting up a complex distributed experimental container composed of nodes and networks; the experimental steps are enacted on relevant parts of the container.

4. Approach

Our approach combines the concepts of experimental containers and interactive notebooks to create a new design pattern that divides the code blocks in a notebook into three categories according to the role they play in the experiment: *container blocks* are devoted to the creation and configuration of experimental container, *execution blocks* perform a series of experiment steps within that container, and *presentation blocks* analyze and visualize results. Figure 1 shows a simple example where a notebook is split into three sections, each containing one category of blocks.

Container blocks contain instructions that construct the experimental container entities on the target platform. Because this involves performing actions on behalf of the end-user, integration with the platform via an authenticated API is critical. Elements of the experimental container can be created in a number of ways: via an orchestration template packaged alongside the notebook, or via an imperative approach that iteratively builds up the container environment with a series of API calls (e.g., provisioning a node with a disk image, assigning an IP address, etc.). Whatever the mechanism, we end up with an executable process that provisions an experimental container, consisting of all required infrastructure and software, on a shared platform.

Each interactive entity in the experimental container can be given a name e.g., a bare metal node may be designated as a *server*. An experimental environment can be thus

expressed as a set of named entities. Once an experimental container is deployed, we can create a *binding* from the named container entities (e.g., our *server*) to their concrete representations (i.e., a deployed instance designated as the server) such that a remote connection can be established. Specifically, if a server in our container was provisioned as a virtual machine with the IP address 10.100.0.2 and some public key allowing access via SSH, the binding would contain an entry for the server indicating its connectivity via SSH using the provisioned address and key allowing the system to transparently establish a connection to the named entity.

Execution blocks contain the atomic steps of an experiment. Steps in an experiment operate against named entities; the user specifies which to which name a particular block belongs. The *binding* then allows mapping execution requests from the bloc to a corresponding remote entity. A notebook's separation of front-end presentation (the text that is displayed/edited) and back-end execution (where that text is executed as code) allows this to be transparent from the user's perspective and effectively re-purposes the interactive notebook as a terminal multiplexer. Importantly, the indirection of the *binding* makes it possible to change the container's representation (e.g., running on a different platform, in a different topology, or simply a different disk image) and then execute the exact same experimental process as before within the new container, facilitating introducing variation.

Presentation blocks are dedicated to translating any raw output of the experiment to a reader-friendly form, such as a table, graph, or interactive visualization.

This approach plays to the strengths of the notebook, notably its expressivity. All aspects of experimentation happen in partnership with the system, not in accordance with it. Perhaps most beneficially, this approach does not require material changes to existing notebooks, platforms or, arguably, workflows.

5. Implementation

We implemented this design pattern by extending various parts of the Jupyter Notebook infrastructure, namely the JupyterLab [47] front end application, the IPython [48] back end kernel, and the JupyterHub [49] multi-user environment. We deployed a JupyterHub installation that any Chameleon user could access via their existing credentials; upon login, a JupyterLab application exclusively for the user is automatically provisioned within a Docker container and a special access token that provides authentication to Chameleon's API on behalf of the user is implicitly bound to the application. Crucially, any work that a user does within the application is persisted in a home directory backed by a Docker volume associated with the user: this allows the user to come back weeks later and continue their work. Chameleon's object store is additionally integrated as an additional drive in the JupyterLab application, allowing users to archive or access notebooks and other artifacts within the same interface.

Jupyter Notebooks are by default constrained to sharing one back end kernel; to allow us to express the different categories of code blocks our approach requires, we implemented a kernel that is capable of spawning and proxying requests to various remote or local child kernels, similar to prior work on polyglot [50] and reproducible [51] notebooks.

Container blocks: Chameleon, like commercial clouds, exposes several APIs that enable applications to perform actions on behalf of users. For example, all of Chameleon's subsystems have command-line-interface clients that users can invoke with their credentials to e.g., reserve or provision a particular type of bare metal node as if the user interfaced with the Chameleon GUI. To leverage this, *container blocks* execute within a default kernel that has all relevant clients already installed; these kernels support either scripting either with Python or Bash. Within these blocks, users can specify how their environment is created, e.g., by declaring and uploading an orchestration template to Chameleon's orchestration service, or by specifying a series of imperative steps that iteratively build up the environment. Additional Chameleon-specific "building blocks" like Bash scripts and Python libraries are available within the kernel's execution environment to make common tasks (such as provisioning a bare metal node with a certain disk image, assigning a public IP address and then waiting until the SSH service is available) simpler to express.

Transparently storing and utilizing the user's credentials becomes particularly important here. It allows the blocks defining the experimental container to be credential-less, a property important both for security (credentials are not explicitly handled within the notebook) and for shareability (different users will use different credentials); the notebook always provisions the experimental container on behalf of the user. We accomplish this by storing the user's authenticating information in environment variables, a commonly-supported affordance in API clients (Chameleon being no exception.)

Execution blocks: the user can designate which named entity in their environment should execute any particular *execution block*. Code execution requests are then marshalled to their corresponding remote entity and any output is streamed back to the notebook. Traditionally, Jupyter notebook kernels execute code within a local Python REPL (Read-Eval-Print Loop), which interprets Python scripts inputted by the user within a persistent session, allowing e.g., variable assignments that can be referenced in following scripts. To get around the limits of this local execution model, which is bounded by the capabilities of the host running the Jupyter interface, projects such as Pangeo [18] and Jupyter Enterprise Gateway [52] instead proxy code to dynamically-provisioned remote kernels. Similarly, our kernel proxies code execution to a shell session within the user's experimental container, effectively

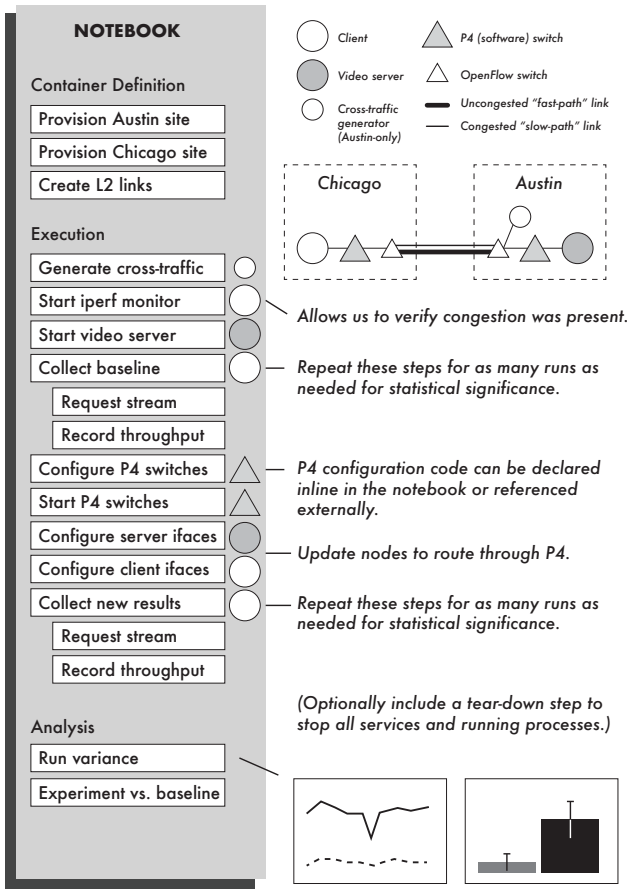

Figure 2. The process of utilizing the approach for an existing complex experiment. The diagram at the top-right represents the experimental container.

giving users the same level of access they would enjoy by logging in via a terminal. The kernel's remote shell session can be initialized transparently from the user's point of view: while a simple solution requires the user to input the *binding* e.g., remote address and authentication information for the named entities, a better solution automatically infers this from the experimental container.

6. Use Case Scenario

To evaluate how well our approach works in practice we applied it to an actual experiment enacted on the Chameleon testbed [53]—a screencast of this exploration can be viewed online [54]. This experiment seeks to answer the question whether performance of adaptive bitrate streaming can benefit from traffic engineering based on HTTP headers, where retransmit requests are routed through a less-congested path. After provisioning an experimental container consisting of a client, server, and switches connected by two layer-2 links, the experimenter must configure and reset the switches, then start a script that generates congestion on one link, then start the streaming server and instruct the client at the opposite end to request a stream, measuring the total time and average throughput rate. Given the interplay of multiple resources and actions, the experiment is of non-trivial complexity and makes it a good candidate to exercise the features of our approach. We describe below first how the author of the experiment can use our approach to develop the experiment and produce shareable experimental artifacts, and then how these artifacts can be used in repeating, replicating, or producing variations on the original experiment.

6.1. Experiment Author Workflow

The objective of this workflow is to both support the creative process, which is often meandering and unpredictable, but at the same time produce digital artifacts that will allow the experiment author to share the end result of a finished experiment.

1. Foundation: All of the resources required for the experiment, including bare metal nodes at two geographic sites, isolated layer-2 networks between these sites, and OpenFlow-enabled [55] routing across those links, can be provisioned by Chameleon. The container creation is expressed as a series of *container blocks* throughout the notebook containing instructions that invoke Chameleon's APIs to reserve and provision resources on the testbed. Importantly, the author knows that any future Chameleon user will be able to execute these blocks without modification, due to the commonality of the platform (i.e., testbed) and the transparent utilization of the future user's Chameleon authentication credentials.

2. Iteration: Every aspect of the experiment can be built up iteratively. While the experimenter may start by provisioning a simple environment, entities can be added to or removed from the container declaration gradually via *container blocks*. Steps in the experiment process are similarly expressed as *execution blocks*, which can be added, removed, or adjusted inline quickly as the understanding of the experiment changes. The experimenter assigns each *execution block* to a particular entity in the experiment; for example, Figure 2 shows how the first step of generating congestion is performed on a dedicated node. Declaring this step in a block allows the experimenter to quickly adjust the level of congestion and see its relationship to the results. There is flexibility in what constitutes an *execution block*: if the experimenter wishes to directly install or configure software environments (e.g., the switches in this example) within their container outside of the notebook, these changes can be persisted as a snapshot; in this case the pre-configured snapshot can be launched in an *container block*, allowing the experimenter to return weeks later and be able to re-provision the exact same experimental container.

3. Publication: Any artifacts intrinsic to the experiment such as disk images can be published as public artifacts to Chameleon's image repository for other users to access. Results can be plotted as graphs or represented as images

suitable for documents or slides. The notebook along with any processed data or code referenced therein represents a single reproducibility artifact that can accompany a conference paper—and can also be shared via Chameleon's object store as described in the implementation section.

6.2. Repeating/Replicating Workflow

In this workflow we seek to demonstrate how the produced digital artifacts can be used to repeat or replicate the experiment.

- **Repeatability**: The original author can run their entire experiment multiple times using the experimental artifacts to collect additional samples or data points; we were able to repeat this experiment via the notebook at different times establishing the experimental environment without problems.
- **Replicability**: A paper reviewer can re-run the notebook from the top down, provisioning the same experimental container and repeating the same process against that container, ultimately achieving the same or similar results; we simulated this situation by asking colleagues, some with little or no expertise in the subject matter to rerun this experiment which they were able to accomplish easily.
- **Environment variation**: The experimental container can be redefined to inject variation; we were able to provision the container on a different set of bare metal nodes, as the nodes used in the original experiment had been reserved by another user.
- **Process variation**: The same environment can be used, but the process can be altered; we could easily change the level of congestion generated on the link to see how it impacted the results.
- **Analysis variation**: The same environment and process can be used, but the results can be processed in a different manner, compared against a different baseline, or contain more samples; we re-organized the analysis code to better understand its mechanisms and formatted the graph differently.

This design supports not only repeatability (from the author's perspective), replicability (from another's perspective), but also provides a flexible model for variability, offering a holistic solution to the challenges of reproducibility.

7. Conclusions

While there is general agreement on the importance of repeating and replicating research, the cost of doing that is often prohibitive, especially as it often comes at the expense of doing more research. This is particularly difficult for computational experiments that are complex, and where variation often depends on selecting and configuring a complex environment.

We made the case for extending the concept of computational notebooks, as captured in the Jupyter notebook implementation, with complex experimental containers, as implemented in the Chameleon cloud, and introduced a design pattern in which the experimenter uses notebooks to develop the experimental environment as well as the process. We demonstrate by means of a complex experimental scenario how this process works in practice both from the perspective of facilitating the original experiment development and of repeating it and introducing variations, greatly simplifying both. Our demonstration shows that using the techniques introduced here even complex experiments can be developed from the ground up to be replicable, facilitating both the work of the experiment author and their sharing with wider community.

Acknowledgments

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.

References

[1] D. G. Feitelson, "From repeatability to reproducibility and corroboration," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 3–11, 2015.

[2] B. Abbott, S. Jawahar, N. Lockerbie, and K. Tokmakov, "LIGO scientific collaboration and virgo collaboration (2016) gw150914: first results from the search for binary black hole coalescence with Advanced LIGO. physical review d, 93 (12). issn 1550-2368," *PHYSICAL REVIEW D Phys Rev D*, vol. 93, p. 122003, 2016.

[3] Amazon Web Services. [Online]. Available: https://aws.amazon.com/

[4] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth, "Chameleon: a scalable production testbed for computer science research," in *Contemporary High Performance Computing: From Petascale toward Exascale*, 1st ed., ser. Chapman & Hall/CRC Computational Science, J. Vetter, Ed. Boca Raton, FL: CRC Press, May 2019, vol. 3, ch. 5, pp. 123–148.

[5] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 03, pp. 80–85, May 2012.

[6] Project Jupyter. [Online]. Available: https://jupyter.org/

[7] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[8] S. Wolfram, "The mathematica book," *Assembly Automation*, 1999.

[9] F. Pérez and B. E. Granger, "Ipython: a system for interactive scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 21–29, 2007.

[10] Code Ocean. [Online]. Available: https://codeocean.com/

[11] A. Brinckman, K. Chard, N. Gaffney, M. Hategan, M. B. Jones, K. Kowalik, S. Kulasekaran, B. Ludäscher, B. D. Mecum, J. Nabrzyski *et al.*, "Computing environments for reproducibility: Capturing the Whole Tale," *Future Generation Computer Systems*, vol. 94, pp. 854–867, 2019.

[12] Nextjournal. [Online]. Available: https://nextjournal.com

[13] Binder. [Online]. Available: https://mybinder.org/

[14] Wolfram Cloud. [Online]. Available: https://www.wolframcloud.com/

[15] Azure Notebooks. [Online]. Available: https://notebooks.azure.com/

[16] Amazon EMR. [Online]. Available: https://aws.amazon.com/emr/

[17] Google Colab. [Online]. Available: https://colab.research.google.com

[18] A. A. Arendt, J. Hamman, M. Rocklin, A. Tan, D. R. Fatland, J. Joughin, E. D. Gutmann, L. Setiawan, and S. T. Henderson, "Pangeo: Community tools for analysis of earth science data in the cloud," in *AGU Fall Meeting Abstracts*, 2018.

[19] S. R. Piccolo and M. B. Frampton, "Tools and techniques for computational reproducibility," *GigaScience*, vol. 5, no. 1, p. 30, 2016.

[20] 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*. ACM, 2018, p. 32.

[21] B. M. Randles, I. V. Pasquetto, M. S. Golshan, and C. L. Borgman, "Using the Jupyter notebook as a tool for open science: An empirical study," in *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 2017, pp. 1–2.

[22] M. Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonnier, "The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication." in *AGU Fall Meeting Abstracts*, 2014.

[23] B. Grüning, J. Chilton, J. Köster, R. Dale, N. Soranzo, M. van den Beek, J. Goecks, R. Backofen, A. Nekrutenko, and J. Taylor, "Practical computational reproducibility in the life sciences," *Cell systems*, vol. 6, no. 6, pp. 631–635, 2018.

[24] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005.

[25] J. Turnbull, *The Docker Book: Containerization is the New Virtualization*. James Turnbull, 2014.

[26] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The popper convention: Making reproducible systems evaluation practical," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2017, pp. 1561–1570.

[27] A. Davison, "Automated capture of experiment context for easier reproducibility in computational research," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 48–56, 2012.

[28] M. A. Sevilla and C. Maltzahn, "Popper pitfalls: Experiences following a reproducibility convention," in *Proceedings of the First International Workshop on Practical Reproducible Evaluation of Computer Systems*. ACM, 2018, p. 4.

[29] A. David, M. Souppe, I. Jimenez, K. Obraczka, S. Mansfield, K. Veenstra, and C. Maltzahn, "Reproducible computer network experiments: A case study using popper," in *Proceedings of the 2nd International Workshop on Practical Reproducible Evaluation of Computer Systems*. ACM, 2019, pp. 29–34.

[30] K. Keahey, P. Riteau, J. Anderson, and Z. Zhen, "Managing Allocatable Resources," in *Proceedings of The IEEE International Conference on Cloud Computing (CLOUD 2019)*. IEEE Press, 2019.

[31] R. McGeer, M. Berman, C. Elliott, and R. Ricci, *The GENI book*. Springer, 2016.

[32] K. Keahey and T. Freeman, "Contextualization: Providing one-click virtual clusters," in *2008 IEEE Fourth International Conference on eScience*. IEEE, 2008, pp. 301–308.

[33] E. National Academies of Sciences and Medicine, *Reproducibility and Replicability in Science*. Washington, DC: The National Academies Press, 2019. [Online]. Available: https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science

[34] C. Drummond, "Replicability is not reproducibility: nor is it good science," 2009.

[35] Blazar. [Online]. Available: https://docs.openstack.org/blazar/

[36] Heat. [Online]. Available: https://docs.openstack.org/heat/

[37] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An Overlay Testbed for Broad-coverage Services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, Jul. 2003. [Online]. Available: http://doi.acm.org/10.1145/956993.956995

[38] COSMOS: Cloud Enhanced Open Software Defined Mobile Wireless Testbed for City-Scale Deployment. [Online]. Available: https://cosmos-lab.org/

[39] POWDER: Platform for Open Wireless Data-driven Experimental Research. [Online]. Available: https://powderwireless.net/

[40] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. I. Ivanov, M. van Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[41] R. Ricci, E. Eide, and C. Team, "Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications," *; login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

[42] Introducing five new Amazon EC2 bare metal instances. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/02/introducing-five-new-amazon-ec2-bare-metal-instances/

[43] AWS DirectConnect. [Online]. Available: https://aws.amazon.com/directconnect/

[44] ExpressConnect. [Online]. Available: https://www.alibabacloud.com/products/express-connect

[45] Y.-T. Chang, R. T. Hood, H. Jin, S. W. Heistand, S. H. Cheung, M. J. Djomehri, G. Jost, and D. S. Kokron, "Evaluating the suitability of commercial clouds for NASA's high performance computing applications: A trade study," 2018.

[46] CloudFormation. [Online]. Available: https://aws.amazon.com/cloudformation/

[47] JupyterLab. [Online]. Available: https://jupyterlab.readthedocs.io/en/stable/

[48] IPython. [Online]. Available: https://ipython.org/

[49] JupyterHub. [Online]. Available: https://jupyterhub.readthedocs.io/en/stable/

[50] G. Wang, M. C. Leong, and B. Peng, "Script of scripts," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC18 Poster). IEEE Press*, 2018.

[51] E. Bavier, L. Courtès, P. Garlick, P. Prins, and R. Wurmus, "Guix-hpc activity report 2017–2018," 2019.

[52] Jupyter Enterprise Gateway. [Online]. Available: https://jupyter-enterprise-gateway.readthedocs.io/en/latest/

[53] D. Bhat, J. Anderson, P. Ruth, M. Zink, and K. Keahey, "Application-based QoE Support with P4 and OpenFlow," in *Proceedings of the IEEE Conference on Computer and Networking Experimental Research using Testbeds (CNERT 2019)*. IEEE Press, 2019.

[54] J. Anderson, "Reproducibility by Default: Jupyter on Chameleon (screencast)," Oct. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3489724

[55] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.