KBase: A Platform for Reproducible Bioinformatics Research

William J. Riehl*, Shane Canon*, Jay R. Bolton*, Boris Sadkhin†,
Gavin Price*, Paramvir Dehal*, Tianhao Gu†, Michael Sneddon‡ and Roman Sutormin§
* *Lawrence Berkeley National Laboratory, Berkeley, CA USA,*
† *Argonne National Laboratory, Lemont, IL*
‡ *Zymergen, Inc., Emeryville, CA*
§ *Google, Inc*
Email: See https://kbase.us for contact information

***Abstract*—Reproducibility is a core tenet of the scientific process, yet it remains elusive for much of the sophisticated analysis required in modern science. In this paper we describe how reproducibility is addressed in the KBase platform, a web-based platform for performing sophisticated analysis of biological data with the goal of enabling reproducible, predictive biology. We give an overview of the architecture and some of the key design considerations. Containers play a key role in the KBase design and how it achieves a measure of strong reproducibility. We explain how containers are utilized in the platform and some of the additional considerations that aid in the goal for reproducibility. Finally, we compare KBase with other similar platforms and systems.**

1. Introduction

Reproducibility is at the foundation of the scientific method, but achieving reproducibility in modern scientific analysis can be challenging. This is especially true in areas like computational biology, where a wide-range of tools and complex workflows are often required to carry out even the most routine analyses. The Department of Energy's Systems Biology Knowledgebase (KBase) [1] is a platform that has been designed to allow scientists to conduct reproducible analysis that can easily be shared with collaborators and the broader community. Containers are at the heart of the design that enables this reproducible analysis. In this paper, we will describe some of the challenges in achieving reproducibility in computational biology and explain how the KBase platform addresses many of those challenges. We will give an overview of the KBase architecture for executing analyses and the role of containers in this design. We will also discuss related works and future plans for KBase related to containers.

2. Problem Statement

Enabling reproducible analysis in biology is challenging for several reason. Analysis typically relies on precise versions of tools or libraries, which can be difficult for a user trying to reproduce some analysis to determine and gather. These tools may even have conflicting dependencies making it difficult to construct a single environment to execute all steps. In addition, these tools may need to be run in a specific sequence to reproduce a result and tracking and determining what steps were done to reach a result requires careful record keeping. Each of these factors contribute to making it difficult for a scientist to repeat their own analysis later in time, and even more so for someone reading a paper and attempting to reproduce a result. As tools and systems are updated and changed, this problem gets harder.

3. Design

KBase is a web-based platform that enables users to perform reproducible analysis that can be easily shared with others. The platform consists of a set of core services that provide data storage, execution services, user interfaces, user management, and other basic services. The platform is designed to be extensible via an SDK that allows new functions to be dynamically added to the system. The platform also features an advanced data system that handles storing data, managing access, and tracking relationships and provenance.

Two key design principles influence the architecture of KBase.

- Enable reproducibility by encapsulating analysis and capturing provenance on all data in the system.
- Provide an extensible platform that supports the addition of new data types and applications without changes to the core platform.

Several KBase components play a role in enabling these principles. Here, we focus on the software developer kit and execution engine, but other components such as KBase's data services play an equally important role.

3.1. User Experience

Users interact with KBase primarily through a web-based user interface built on Jupyter. Users generally start by uploading their data into the KBase platform and transforming it into a data objects that get stored in the KBase data

978-1-7281-6028-3/19/$31.00 ©2019 IEEE
DOI 10.1109/CANOPIE-HPC49598.2019.00009

store. Data can be uploaded via a drag and drop interface or via a Globus Endpoint [2]. All objects are tagged with metadata including information about its provenance. This provenance is used to keep track data's path through the system along with any transformations made to it by running applications. Each time a KBase function is executed, any generated output objects will include the name of the function, its version, input objects and parameters in its provenance. This allows a user to trace up the provenance chain of an object to see what sequence of actions led to its creation.

Once data is uploaded, users can run a wide range of applications (apps) that do either analysis or transformations to the user's data. These apps can range in complexity from simple visualization tools, to a multi-step analytical software requiring terabytes of reference data and large HPC resources.

3.2. KBase Software Developer Kit and App Catalog

The KBase SDK is designed to enable third-party developers outside the KBase project to easily add new functionality into the system while constraining the behavior to enforce KBase's guiding principles. A developer can use the SDK to add an app into KBase that can then be executed in the KBase framework, taking advantage of the available data and computational resources. Figure 1 illustrates the flow and how the various components in the KBase architecture interact. The developer uses the SDK to generate a set of template files that they can then modify to add the dependencies and logic specific to their app. The SDK also provides tools to simplify interacting with the KBase data stores and local file storage to download data for the applications to operate on and store results. Along with the KBase-specific tools, other apps and functionality can be called using the SDK. These are effectively installed as dependencies, and called using the subjob call mechanism described in detail below. This allows developers to better decompose their code and make common tools more available throughout the system. Finally, the KBase SDK provides tooling to build a Docker image that encapsulates the developer's app and all requirements and tools to do local testing outside of KBase. Since everything is containerized, the app developer can be confident that if the module works in local testing, it will have the same behavior once it is registered in KBase.

Many modules follow a "wrapper" style use of the SDK. The goal here is to add the functionality of existing software to make available through KBase. For example, MEGAHIT [3] is a software package that assembles metagenomic read data into contig sequences that can later be annotated and analyzed for their biological capabilities and significance. In essence, it takes in a set of reads (a Reads object), computes on it, and produces an assembly (an Assembly object). This complex command-line tool is wrapped into a KBase module using the SDK. The execution flow of the module code works as follows:

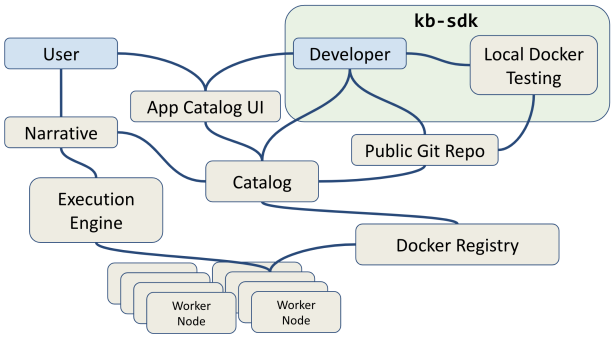

Figure 1. Diagram of the key components used for developing and executing applications in KBase.

1) An SDK utility function downloads the Reads object into the scratch area as a file.
2) The wrapper function crafts and executes a command line call to execute MEGAHIT using the downloaded file, along with other user-input parameters.
3) A second SDK utility uploads the generated assembly file as an Assembly object.
4) Another SDK-generated module is run on the new assembly to build a visualization that describes the quality of the result.
5) The resulting visualization (an HTML file and attached Javascript code) and other information is bundled together into a Report that gets saved into KBase.
6) These objects are all returned as the result of the job.

The resulting SDK module, therefore, is a combination of a wrapper around a command line program, and several KBase utility functions that enable this functionality in KBase.

Once the developer has wrapped an application and implemented any logic to marshal data, the developer checks their code into a public Git-based repository such as GitHub, and then registers the public repository in the KBase App Catalog. The App Catalog, amongst other things, keeps track of all the apps registered in the system, their release state, and metrics on their execution. During registration, the App Catalog builds a Docker image using the Dockerfile contained in the repository and then saves the image into a private KBase Docker registry, versioning it with the git commit id. The Catalog service can also associate a semantic version tag with the image and manage the process of promoting a version from development to beta and to release. Applications in development and beta have limited visibility to encourage users to only use them for testing, while released versions are clearly visible to all users.

During registration the Catalog service performs the following sequence of actions.

1) Performs a local git checkout of the repository

2) Reads a metadata file included in the repository to determine the name of the module and other details
3) Builds the docker image using the included Dockerfile
4) Initializes reference data if required (see below)
5) Executes the new container in a special report mode to ensure minimal functionality
6) Pushes the image into a local Docker registry
7) Saves metadata and other information about the module in the app catalog

During registration, the developer can see the log of the entire process in case any steps fail. Once the application is registered, it is immediately visible in the Catalog and can be executed in KBase.

3.3. Execution Services

An end user triggers app execution through the KBase Narrative interface, a web-based graphical interface based on the Jupyter Notebook [4]. The Narrative interface generates the form that guides the user through the process of entering all of the inputs and parameters required for app execution. All of these inputs are defined by the developer in the Git repository and validated between the Narrative interface and the SDK. Once a user has entered all of the required information and clicks the run button, the Narrative sends that information to the KBase execution engine. This service validates and authenticates the request and submits the job to an HTCondor Batch scheduler [5]. HTCondor schedules the job on one of a pool of workers, choosing a worker based on the application's requirements (e.g. memory and processors). When the job starts, a parent job runner process begins that translates the request into a Docker container execution. The job runner reads the app id, function name, and version, and uses these to query the Catalog for the correct Docker image. The parameters for execution are written into a file in a scratch area that is mounted into the Docker container for use as temporary and intermediate file storage. Once the container is started, the SDK-generated entry point uses that request to call the appropriate function in the SDK module. At this stage, the requested command is run inside the container, and can make use of the scratch disk area for local storage. In the end, any generated data objects are uploaded back into the KBase data stores.

The SDK also helps to serialize the output into a file in the scratch area, that is then read by the job runner when the application exits. This result is returned back to the execution engine where it is saved and returned back to the Narrative for the user to view. The runner also monitors the console output from the container and streams those logs back to the execution engine, so that the Narrative can display these logs to the user. These logs are useful for tracking the progress of an application's execution and diagnosing any errors. The logs remain stored and available to users and administrators for diagnostic purposes.

To ensure reproducibility, state is not shared from run to run. Each job run starts a new SDK-based container with

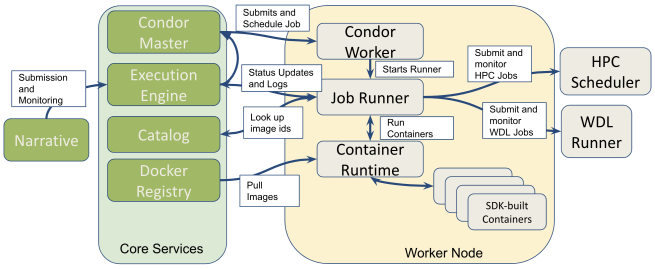

Figure 2. Diagram of the execution flow for an application.

a freshly initialized scratch space that is cleared at the end of the run. As noted previously, the Docker image used for the execution is tied to the semantic version and Git tag of the repository used to build the image, and that image is maintained in a private Docker registry. Any data generated by an application and saved to the KBase data store includes the application name, version, input data, and parameters in order to track the provenance of each data object. This allows the user to understand how data was generated and reproduce the execution later. This all ensures that each job run remains independent of other runs, but can be reproduced by running the app again with the same version and the same inputs. After the run, runtime performance statistics, including run length and success state, are uploaded to the Catalog service for metric tracking. Fig. 2 illustrates these execution interactions for a job.

3.4. Subjobs

Many KBase applications need to share or reuse logic, especially related to data access. Rather than having the developers replicate the code needed for common tasks within their own modules, the KBase job runner provides a subjob call mechanism. This allows an SDK module to request the execution of a function provided by another module. For example, an assembly application which assembles raw genomic reads into assembled contigs will need to contain the logic for loading raw reads from the KBase data store, do the analysis, generate a report about the quality of the assembly, and then save the newly assembled contigs into the datastore. Rather than replicating the logic needed to access, retrieve and download the raw reads, a developer can use existing utility apps to do this. Likewise, instead of installing a quality control report app, such as FastQC, into the assembly application, the app can leverage an existing app to perform that step.

Commonly used utility apps are used to stage, transform, and upload data. The runner automatically mounts the scratch area for the calling job into the subjob's container, so it becomes a shared storage space. This avoids any additional copying or transferring of the data between the main job and subjobs. Likewise, any data manipulated on disk by a subjob becomes available to the calling job. The job runner also keeps track of all the subjobs and parameters used during the lifetime of a job, and this information can

be retrieved and included in the provenance for any data that is ultimately saved in KBase. In all, calling a subjob works in a manner very similar to calling the original job, and makes use of the same resources.

The subjob mechanism was originally developed to simplify interacting with the KBase object model, which can be complicated to learn. However, it has turned out to be useful in also building composable workflows. Using subjobs, individual modules can encapsulate certain common functions through a workflow app, which composes those individual steps via the subjob call mechanism. One example of this composition would be a pipeline where a master app would call several apps in succession to operate on the results of the previous step. Those sub apps themselves could then call others as needed. In addition to supporting on-node parallelism, the workflow job can also submit child jobs through the execution engine for multi-node parallel execution.

3.5. Reference Data

Many bioinformatics tools rely on curated reference data or other supporting data. For example, a genome data filtering tool may have a database of common contaminants. These datasets can be tens or hundreds of gigabytes in size, which is impractical for storage in app images. To address this, the SDK supports versioned reference data collections. The SDK metadata includes a semantic version tag for the reference data that it requires. During module registration, the system checks if the required version of the reference data has been loaded. If not, the registration service calls a special initiation step for the module. It runs the container and volume mounts the area where the reference data will be stored in the system. A script provided by the module developer can download the data from an external location and can perform any other steps needed to initialize the data for later use. The prepared data is then saved in the volume mounted location. Once this process is completed successfully, the script notifies the registration process that reference data creation is complete, and the data collection is versioned. This volume is only writable during registration phase, and is read-only during app execution in order to prevent accidental modification. During execution, only the specified versioned of the data is mounted to prevent tools from inadvertently picking up the wrong version.

3.6. Support for HPC applications

The majority of the applications in KBase are designed to run on a single node or use simple parallel execution (e.g. pleasantly parallel). However, there are a growing set of applications in the bioinformatics space that can take advantages of HPC-class systems by using Message Passing Interface (MPI) or languages like Unified Parallel C (UPC). To support these applications, KBase can route HPC enabled jobs to HPC resources (currently only at NERSC, the National Energy Research Scientific Computing Center). A standard HTCondor worker runs on a service node associated with the HPC system. Typically only HPC enabled jobs are routed to these workers. During execution, the job runner will start the application as normal, performing the validation and data staging steps. The data is staged into the HPC systems parallel file system such as the Lustre based scratch file system at NERSC. Once the steps that don't benefit from parallel execution are complete, the SDK application code can generate a submit script for the local batch system (e.g. SLURM at NERSC), and then make a special callout to the job runner to submit the job to the HPC scheduler. The submission occurs outside of the container to avoid the need for the SDK image to include any specific batch system clients (see Figure 2 for the point in the execution flow where HPC components can be invoked). Since the data is already staged in the parallel file system, the HPC application being invoked by the submission script doesn't require any specialization for KBase and would typically not need to communicate with KBase web services which could introduce inefficiencies. The job runner will monitor the progress of the batch job through its execution. Once the SDK app detects that the HPC jobs has completed, it can perform any required post processing steps. This would typically include uploading any generated data products into the KBase data store. These steps can also use the subjob mechanism like regular non-HPC applications. Since HPC sites do not typically allow regular users to run or use Docker, the job runner can be configured to use HPC Container Runtimes instead. Currently only Shifter [6] is supported since it is available at NERSC, but the runner has an extensible design, so support for other HPC container runtimes such as Singularity [7] or Charliecloud [8] can easily by added.

3.6.1. Workflow Language Support. The bioinformatics space has recently seen a growth in workflow tools that utilize standardized descriptions. For example, the Common Workflow Language [9] and Workflow Description Language (WDL) [1] are two examples of standardized descriptions for workflows that then can be implemented by a tool developer. Both of these standards have native support for specifying a container image as part of the workflow description. Given their flexibility and growing community adoption, the KBase team felt it was important to support these models. The challenge is marrying a tool that is primarily file-based with KBase's object based model.

For the first implementation of this support, KBase is using an approach similar to what is done in the HPC model described above. A standard SDK app can be created in which the application performs validation and data staging using native KBase modules. The app then generates the input files and workflow description files and calls out to the job runner to launch the workflow. KBase currently uses Cromwell [2] to execute these workflows. Cromwell was chosen because it offers support for both CWL and

1. https://software.broadinstitute.org/wdl/documentation/
2. https://cromwell.readthedocs.io

WDL and has been adopted by the Joint Genome Institute, a close partner of KBase. Similar to the HPC model described above, when submitting a job that uses a workflow language, the steps up until the Job Runner remain the same. Instead of instantiating a Docker container, however, the job runner calls out to Cromwell to start the job using the specified workflow document. The runner then monitors the execution of the Cromwell job through completion. Once complete, the SDK module can do KBase specific post processing steps like storing the output and generating reports. It then returns the resulting information to the Narrative interface, keeping the user experience the same, regardless of the implementation details of the app. These additional steps help bridge the gap between the file-based WDL and object-based KBase data store.

While these workflow specifications provide a standardized way to capture a workflow, they do allow a level of flexibility that could break some of KBase's principles around reproducibility. For example, a valid workflow can specify soft tags like "latest" for its image tag which would likely change over time. These standards also do not offer a well defined way for to handle reference data. Consequently, KBase must review any apps using WDL to ensure that the generated workflows adhere to the KBase principles.

4. Related Work and Discussion

The KBase project has many goals related to biosciences and computing. These include giving users the ability to freely upload and analyze their data, share the results of their analysis along with any narrative documentation, and eventually the ability to relate various data objects to each other in a meta-analysis that can lead to novel predictions of biological significance. The primary computing challenge that KBase faces revolves around giving users and developers the freedom to analyze their data using tools of their choice. That is, users should be able to bring the computational tools they want to use to their data in KBase, and apply them there. Once these tools have been applied, the results should be stored in a way that make them easy to (1) reproduce those analyses, and (2) compare the results to each other. That implies storing all information necessary to repeat the computational jobs, and storing the results in a format-agnostic way.

To approach these goals, KBase makes use of several technologies, both built specifically for the project, and otherwise available to the computing science and bioinformatics communities. To approach the problem of making computational jobs reproducible, KBase makes substantial use of container technology, particularly through Docker images. As described above, this allows KBase to compile and store images that contain all code necessary and sufficient to run an analysis, along with specific version tags that can be used to re-invoke the same tools at a later time. There are many tools and platforms that address some of the motivations and aspirations of the KBase platform, so an exhaustive list is impractical. We will briefly discuss a few common examples, to illustrate the overlap and differences. The workflow space is particularly broad with a rich set of tools. CWL and WDL, already discussed, provide a standardized description of a computational workflow that can be implemented independently, and can be coupled with an execution engine like Cromwell for running. There are dozens of tools that follow these standards[3]. Many of these build on standard container technologies including Kubernetes and Airflow or integrate with HPC batch schedulers such as SLURM and PBS. As described above, KBase has elected to use one of these existing tools (Cromwell) versus implementing the standard directly into the KBase execution engine.

There are also web-based workflow tools that aim for reproducibility and leverage many of the same technologies as KBase such as containers and Jupyter. Galaxy [10] is a mature project that is popular in the biology space and allows workflows to be graphically constructed and executed. GenePattern [11] is another platform for reproducible bioinformatics research that includes the construction of pipelines and workflows. The chief difference between these platforms is KBase's larger vision of building a platform that allows data from all users in the system to be connected together and drive insight. The KBase vision requires that data stored in the platform contains additional context so that it can be automatically computed on. Many of the tools listed above operate at a file level and maintain minimal provenance metadata about those files.

This work is primarily focused on a discussion of KBase's SDK and execution environment, some of the ways it aims to achieve reproducibility, and the role containers play in this architecture. However, the SDK and the execution environment are just one component in the overall architecture. The data storage components also play a critical role in reproducibility by maintaining provenance about all data in the system and ensuring that any individual with access to a given object can access any other data object that contributed to its creation. We have omitted a deeper discussion on these components. We also note here that container technologies are used throughout the KBase platform. For example, the core services, including the data stores and execution engine, are containerized and managed via Rancher [4]. In the discussion above we focused on using the SDK to describe analysis applications that run for some bounded period of time, but the SDK can also be used to build and package dynamic services. These are semi-persistent microservices that are typically used to provide APIs or backend-for-frontend utilities. For example, a dynamic service exists to serve small pieces of metagenomic data (which typically ranges from gigabytes to terabytes in size) for rapid visualization in a web browser, while another acts as a data search API. These containers, while they persist, are also intended to act as temporary caches, reducing the time necessary to repeatedly access the results of heavy database calls or the intermediate results of long-running compute jobs. Another use of containers is in the Narrative platform. The Jupyter-based Narrative service

3. https://www.commonwl.org/
4. https://rancher.com

starts a container for each individual user to ensure data and privileges cannot leak between users.

5. Future Work

KBase is exploring several enhancements related to the execution environment. For example, we are exploring an interface that would allow users to create custom pipelines of existing apps through the Narrative interface. User-created and curated pipelines will be saved and versioned as well, and can be shared among users. This would allow non-developers to generate workflows and share them with the community. In the future, KBase plans to enable support for users to "Bring your own compute" to the platform. In this model, users can run an agent on local resources or cloud provisioned resources that can be used to offload their jobs. This mode of execution could become important as the KBase user community grows and potentially outpaces our ability to provide resources. Clearly, ensuring reproducibility in this model requires careful thought and consideration.

Another future item is to provide more sophisticated ways to express complex workflows. The recent addition of support for WDL should address many of these use cases, but this support needs to be integrated throughout the system with careful consideration given to the user experience. For example, how should results and output be presented to a user who has run a parallel analysis of thousands of data sets?

There is also a growing set of tools emerging in the container space and Kubernetes ecosystem that could influence the future of KBase. KBase plans to transition to Kubernetes for managing its core services in the near future, but Kubernetes could potentially be utilized for other aspects of the platform such as workflow execution. KBase's execution engine has been specifically designed around the needs of bioinformatics workloads, therefore, finding the appropriate role and model for integration with Kubernetes for workflow execution requires investigation. Kubernetes-based workflow frameworks like Argo could potentially be integrated into the platform either as a substitute for HTCondor or as a separate execution model. For example, similar to how WDL and HPC execution have been enabled, the SDK developer could express complex workflows as an Argo workflow that would be executed inside KBase. Regardless of the approach, careful consideration will be needed to ensure that reproducibility is not sacrificed.

6. Conclusion

The KBase platform strives for strong reproducibility and relies on container technologies to play a key enabling role. Achieving reproducibility requires treating it as a fundamental design principle that is factored into all aspects of the design of the platform; from how the software is packaged and tracked, to how execution takes place and how data is handled. Every step of analysis execution in KBase has been designed and implemented with reproducibility in mind. The ability for containers to encapsulate all of the software and configuration used for execution, as well as provide access control for data has been instrumental in building out this architecture, and making analysis reproducible. Future work will focus on expanding this architecture to better embrace available HPC resources and supporting larger scale analyses against increasingly growing data sets.

Acknowledgments

This work is supported as part of the Genomic Sciences Program Department of Energy Systems Biology KnowledgeBase (KBase) funded by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research under Award Numbers DE-AC02-05CH11231, DE-AC02-06CH11357, DE-AC05-00OR22725, and DE-AC02-98CH10886.

References

[1] A. P. Arkin, R. L. Stevens, R. W. Cottingham, S. Maslov, C. S. Henry, P. Dehal, D. Ware, F. Perez, N. L. Harris, S. Canon *et al.*, "The doe systems biology knowledgebase (kbase)," *BioRxiv*, p. 096354, 2016.

[2] I. Foster and C. Kesselman, "The globus toolkit," *The grid: blueprint for a new computing infrastructure*, pp. 259–278, 1999.

[3] D. Li, C. M. Liu, R. Luo, K. Sadakane, and T. W. Lam, "MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph," *Bioinformatics*, vol. 31, no. 10, pp. 1674–1676, May 2015.

[4] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

[5] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: a distributed job scheduler," in *Beowulf cluster computing with Linux*. MIT press, 2001, pp. 307–350.

[6] D. M. Jacobsen and R. S. Canon, "Contain this, unleashing docker for hpc," *Proceedings of the Cray User Group*, 2015.

[7] G. M. Kurtzer, V. Sochat, and M. W. Bauer, "Singularity: Scientific containers for mobility of compute," *PloS one*, vol. 12, no. 5, p. e0177459, 2017.

[8] R. Priedhorsky and T. Randles, "Charliecloud: Unprivileged containers for user-defined software stacks in hpc," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2017, p. 36.

[9] P. Amstutz, M. R. Crusoe, N. Tijanić, B. Chapman, J. Chilton, M. Heuer, A. Kartashov, D. Leehr, H. Ménager, M. Nedeljkovich *et al.*, "Common workflow language, v1. 0," 2016.

[10] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[11] M. Reich, T. Tabor, T. Liefeld, H. Thorvaldsdóttir, B. Hill, P. Tamayo, and J. P. Mesirov, "The genepattern notebook environment," *Cell systems*, vol. 5, no. 2, pp. 149–151, 2017.