NISTIR 8297

Interoperability of Web Computational Plugins for Large Microscopy Image Analyses

Peter Bajcsy
Nathan Hotaling

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8297

National Institute of Standards and Technology
U.S. Department of Commerce

NISTIR 8297

Interoperability of Web Computational Plugins for Large Microscopy Image Analyses

Peter Bajcsy
Software and Systems Division
Information Technology Laboratory
National Institute of Standards and Technology

Nathan Hotaling
National Center for Advancing Translational Sciences
National Institutes of Health

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8297

March 2020

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Walter Copan, NIST Director and Undersecretary of Commerce for Standards and Technology

National Institute of Standards and Technology Interagency or Internal Report 8297
33 pages (March 2020)

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8297

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and 78 Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems.

Abstract

This document summarizes conclusions from a workshop focused on Interoperability of Web Computational Plugins for Large Microscopy Image Analyses. The workshop conclusions are classified as practical recommendations and agreements on development and future research related to (1) containerization of execution code, (2) data storage, (3) interoperability requirements of workflow engines for running containerized plugins, (4) standard packaging of web user interface modules, and (5) security of container-based distribution.

Key words

Big image analyses; software containers; interoperability of software; microscopy.

Executive Summary

This NIST internal report provides a summary of the workshop on Interoperability of Web Computational Plugins for Large Microscopy Image Analyses. The workshop was held at NIST Gaithersburg, MD, on December 5-6, 2019, and its web page is accessible at this URL[1].

The workshop brought together representatives from 21 institutions spanning research, academic, and industrial communities focusing on big image analyses in computer cloud environments. Such big image analyses are frequently supported by web client-server systems that enable execution of a wide spectrum of algorithms to extract image-based measurements, and perform image classification, object detection, object registration, object tracking, and object recognition. The purpose of this workshop was to discuss the needs for big image analyses in computer cloud environments in terms of software interoperability. Specifically, the workshop targeted bio-medical and bio-materials science applications, current open-source technical solutions for big image analyses, and community-wide research and development (R&D) interests in defining inter-operable algorithmic plugins for web client-server systems designed for big image analyses.

The workshop discussions focused on:

1. Containerization of execution code
2. Data storage
3. Interoperability requirements of workflow engines for running containerized plugins
4. Standard packaging of web user interface modules
5. Security of container-based distribution.

Practical workshop recommendations include:

1. Applying best practices for containerization [1], [2]
2. Leveraging on-going efforts contributing to the definition of the metadata manifest files [3], [4]
3. Using GitHub repositories for storing plugin manifest files and leveraging existing tools for storing execution profile and error log formats [5], [6]
4. Supporting basic central processing unit (CPU) and graphics processing unit (GPU) data types in user interface (UI) modules
5. Following security guidelines [1].

The workshop attendees agreed that a consensus is possible for (1) developing inter-operable mechanisms for launching containers and error handling, (2) having a well-defined application programming interfaces (API) to access a spectrum of file formats, (3) defining inter-operable plugin manifests with parameter and execution information, (4) supporting basic CPU and GPU data types in reusable UI modules, and (5) defining a protocol for signing plugin containers. More discussion is needed for research topics (1) addressing construction of complex user interfaces for collecting parameters, (2) specifying access API for scalable file formats [7] and capturing computational provenance, (3) defining the methods that would assist in finding plugins in distributed repositories and analyze execution log profile, (4) supporting complex data types in UI modules, and (5) scanning containerized software for security purposes in continuous integration/continuous delivery (CI/CD) workflows. Initial code templates have been provided by NIST[2].

[1] https://www.nist.gov/news-events/events/2019/12/interoperability-web-computational-plugins-large-microscopy-image

[2] https://github.com/usnistgov/WIPP-Plugins-base-templates

Contents

1. Background

There is an increasing interest in enabling discoveries from high-throughput and high content microscopy imaging of biological specimens and bio-material structures under a variety of conditions. As automated imaging across multiple dimensions increases its throughput to thousands of images per hour, the computational infrastructure for handling the images has become a major bottleneck. The bottleneck presents challenges that range from transferring data, storing and archiving, annotating, quantifying, and visualizing, to the mechanisms for applying the latest machine learning and artificial intelligence models by non-computational experts from a variety of application domains. These challenges arise due to big image data, complex phenomena to model, and non-trivial computational scalability that accommodates advanced hardware and cutting-edge algorithms. Furthermore, the challenges are amplified by the need to engage a broad community of experts in analyzing complex image content and the need to reproduce discoveries based on image measurements and any decisions derived from these measurements. Such measurements, discoveries and decisions are critical for biological and bio-materials science applications, for instance, quality assurance of stem cell therapies, design of cancer treatments, high throughput screening in drug discovery, and vaccine discoveries from atomic resolution structures of viruses and protein complexes.

To overcome the aforementioned challenges, several research institutions have prototyped web-based systems in order to facilitate access to large image databases and to high performance computing (HPC) and cloud hardware resources. The existing web-based prototype solutions leverage a variety of web technologies on the client side and a spectrum of databases, scientific computational workflow engines, and communication protocols on the server side in order to hide the infrastructure complexity from the domain application experts and make them more productive in conducting research. While the web-based solutions deliver infrastructure capabilities, their capabilities for processing large images remain limited to the computational tools provided by each development team because the development of new tools is web solution specific and a definition of an inter-operable web computational plugin does not exist.

With the increasing popularity of software containers as standardized units for deployment, there is an opportunity for the communities working with large microscopy images to discuss creating inter-operable web computational plugins. These web computational plugins consist of software containers and web user interface (UI) description files to enter parameters needed for the software execution. Each container packages code with all its dependencies and has an entry point for running the computation in any computing environment. Each UI description file contains metadata about the plugin container and the computation parameters. This description file is intended for generating web UI for entering parameters dynamically. Figure 1 illustrates the workshop focus in the context of users, developers, and system administrators using containerized tools and applying them to very large images in computer cluster and cloud environments.

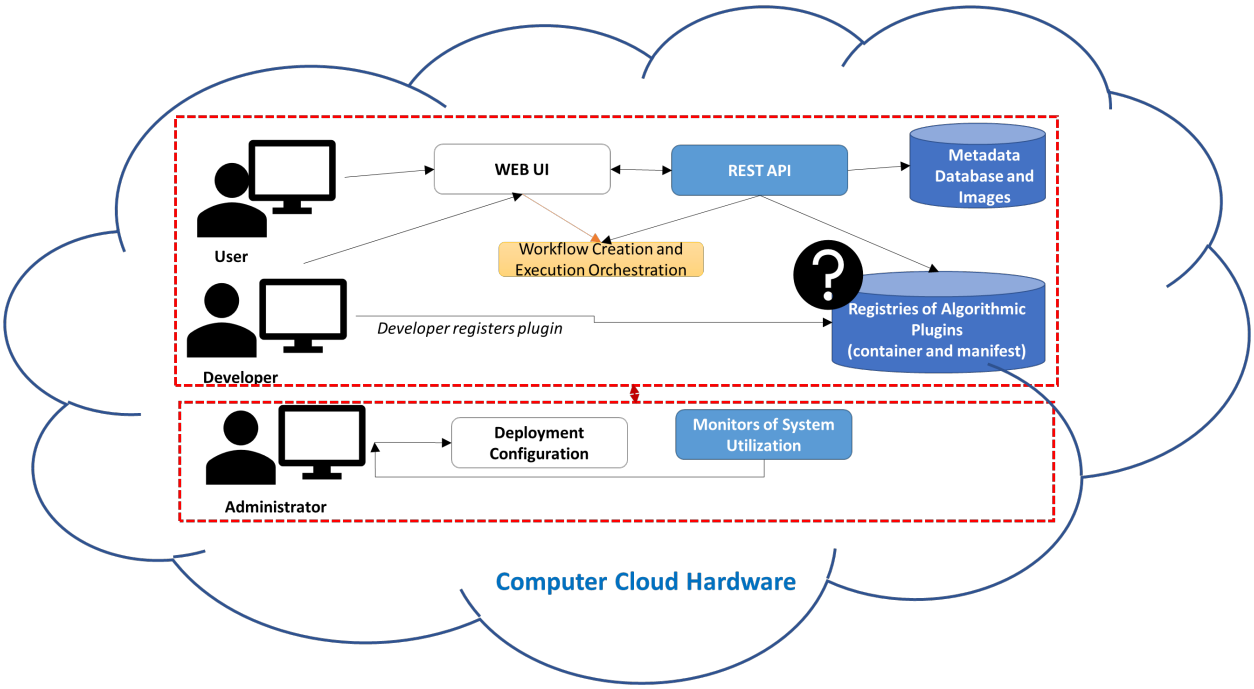

Figure 1: Workshop focus is represented by the question mark in this figure. The focus is viewed in the context of users, developers, and system administrators analyzing very large images in computer cloud environments using a variety of client-server systems. A client-server system follows representational state transfer (REST) application programming interface (API) for creating web services that are interoperable on the Internet.

2. Workshop Description

The workshop was divided into two days based on the focus of each day either on breadth or on depth of relevant topics. During the first day, 15 speakers presented bio-medical microscopy community needs when relying on microscopy image analysis, and several existing solutions and funding mechanisms that are behind advancements of imaging science over large image collections. A subset of presentation slides are available from the workshop web page. The main technologies discussed during the workshop are listed, and links to their descriptions can be found in Appendix A.

In-depth discussions about inter-operable plugins took place in working groups during the second day of the workshop. Five working groups were focused on:

1. Containerization of execution code
2. Data storage and access interfaces for object-, block- and file-level storage
3. Interoperability requirements of workflow engines for running containerized plugins
4. Standard packaging of web UI modules
5. Security of container-based distribution.

Each working group summarized the state-of-the-art and then concentrated on common characteristics of existing solutions and desired best practices. The conclusions of all working groups are provided in this workshop report. There was widespread agreement from attendees that the premise of the conference was valid and the workshop conclusions are highly valuable to the community.

Day 1: Focus on Breadth

During the first day, invited speakers presented a spectrum of viewpoints on how to analyze terabytes (TB) of microscopy images.

1. The speakers represented several funding agencies:
   - National Center for Advancing Translational Sciences (NCATS), National Institutes of Health (NIH)
   - National Cancer Institute (NCI), NIH
   - National Science Foundation (NSF)
   - Center for Devices and Radiological Health (CDRH), United States Food and Drug Administration (FDA)
   - National Institute of Standards and Technology (NIST)
   - Chan Zuckerberg Initiative
2. Non-profit academic and research organizations:
   - National Center for Supercomputing Applications (NCSA), University of Illinois at Urbana-Champaign (UIUC)
   - University of California at Santa Barbara (UCSB)
   - University of Cardiff in Wales
   - KTH Royal Institute of Technology in Sweden
   - University of Dundee in United Kingdom
   - Janelia Research Farm
   - Allen Institute for Cell Science
3. For-profit organizations:
   - Zeiss (or Carl Zeiss AG)
   - PerkinElmer, Inc.

The viewpoints varied across users of microscopes and image analysis software, developers of image analysis software, system administrators of imaging core facilities (software and hardware), and funders of imaging and image analysis activities.

The summary of microscopy image analysis projects that were mentioned during the presentations on Day 1 can be found in Appendix A. The summary is a small sample set of funded efforts devoted to the problem "how to analyze very large microscopy images." These efforts and the team experiences gained while addressing the problem became a starting point for the in-depth discussions on Day 2.

Day 2: Focus on Depth

During the second day, the workshop attendees focused on interoperability of web computational plugins for large microscopy image analyses. The workshop format of Day 2 was based on creating working groups that would enable discussions of current practices and forward-looking standards. The overall goal of all working groups was to determine whether the workshop attendees could reach a consensus on creating inter-operable web computational plugins that can be:

- Chained into scientific workflows/pipelines and
- Executed over large image collections regardless of the cluster/cloud infrastructure components.

Five working groups were formed to understand a variety of technical aspects critical to creating inter-operable web computational plugins. The five technical aspects are:

1. Containerization of execution code
2. Data storage and access interfaces for object-, block- and file-level storage
3. Interoperability requirements of workflow engines for running containerized plugins

4. Standard packaging of web UI modules
5. Security of container-based distribution.

Each working group (WG) had two sessions. Session 1 was focused on a review of existing solutions addressing the WG topic (i.e., what is the current state-of-the-art?). Session 2 was focused on discussing common characteristics of existing solutions and desired best practices to address the WG topic (i.e., how can we work toward a consensus solution?). The working groups 2 and 3 ran in parallel so that the workshop attendees were split, while the working groups 1, 4, and 5 were attended by all interested workshop participants.

3. Conclusions

Containerization of execution code

Review of existing solutions

Containerization of execution code is most frequently performed using Docker and Singularity containers [8], [9] and executed with Kubernetes orchestration middleware [2]. The data access from containers is most frequently achieved via mounting a volume/folder. The content of packaged execution software varies in complexity and represents analytical and simulation functionalities for microscopy imaging.

Common characteristics of existing solutions and desired best practices

Workshop attendees recommended using best practices for containerization of software available on the Web [1]. As a starting point for containerization, the workshop consensus was to recommend using the Alpine Linux container images as the base images, multi-stage builds [10], and continuous integration tools. A consensus is possible for launching algorithms with parameters using the container entry points and error handling for the "run and then destroy"' container scenario. More in-depth discussions are needed for constructing complex user interfaces collecting parameters to container executions. A registry of container images over private and public repositories will be useful to search and find functionalities needed for creating complex image analysis pipelines executed in cloud-based environments [11]. Detailed discussion notes are provided in Appendix B.

Data storage and access interfaces for object-, block- and file-level storage

Review of existing solutions

The most frequently used storage methods for large images are databases and file systems. While file systems are not changing too much, databases are continually being developed. In addition, several new image file formats, such as Zarr, N5, and Apache Arrow are being developed to meet the needs of very large image files (see Glossary). JavaScript Object Notation (JSON) data-interchange format and Extensible Markup Language (XML) format are used frequently for metadata describing inputs and outputs of computational plugins currently. The metadata field definitions and the schema for their storage are currently software/institution/developer dependent.

Common characteristics of existing solutions and desired best practices

The opinions varied when discussing consistent programmatic access to data from containerized computational plugins. To reach a possible consensus will require additional in-depth discussions. While there is a need to build standard interfaces in order to expand the current limited support of mounting folders/volumes on a file system, there is also concern that a standard interface would lock developers into a technology window. The in-depth discussions should also include computational provenance tracking.

Scalable input/output file formats must be considered when data access from containers is achieved by mounting folders/volumes on a file system, Consensus can be reached by specifying access via an application programming interface (API) to a spectrum of file formats. Among the newer file formats suitable to big image analyses, Apache Arrow, Zarr, N5, and tiled Tiff formats are suggested as good candidates for specifying the initial implementations of the access API. There was fairly good consensus/agreement that the API should be the N5 API currently under development once it is able to encapsulate the Zarr specifications (it might be updated to be called Z5 at that point). More in-depth discussions are needed to specify the access API for file formats.

In order to enable chaining of containerized computational plugins, metadata manifest files must be defined. The workshop consensus was to recommend adopting specification, schema, and file type for the metadata manifest files that would come from several on-going parallel efforts, such as OMERO (follow OMERO forums [3]), BisQue (see Appendix A), and Common Workflow Language (CWL) [4]. Detailed discussion notes are provided in Appendix C.

Interoperability requirements for running containerized plugins

Review of existing solutions

A survey of existing solutions among the workshop attendees is provided in Appendix D. In a summary, the most frequent technologies are:

- Workflow editors: Argo and custom solutions,
- Job schedulers: K8 and SLURM (see Glossary),
- Workflow orchestration: Kubernetes [2],
- Workflow representation: scripting files (.py, .js) and Argo file (.yml),
- Workflow monitoring: Kibana + ElasticSearch [5] or none,
- Supported hardware: CPU and GPU platforms,
- Passing parameters from Clients to Servers: JSON files and a list of arguments,
- Finding and fetching containerized steps of workflows: private and public registries,
- Finding and accessing image collections: native file systems,
- Logging errors and warnings: language specific logging and none,
- Target hardware: Commercial Cloud, cluster on premises, and multiple CPU/GPU machines.

Furthermore, the survey included two questions about the current practices. The questions and their answers are provided below:

- What size/volume of data are you expected to process? Giga, Tera, and Peta Bytes, (GB, TB, and PB)
- Is directed acyclic graph (DAG) sufficient to represent workflows? Yes

Common characteristics of existing solutions and desired best practices

As a starting point, the workshop consensus was to recommend using plugin manifest metafiles stored in GitHub and labeled with a tag (e.g., encoded in a file name). More in-depth discussion is needed to define the plugin tags and the methods that would assist in finding plugins.

A consensus was reached that the information in plugin descriptors/manifests must include supported input/output (I/O) file formats and hardware requirements using a predefined schema (JSON or XML). The plugin descriptors should include not only a list of parameters but also the range of valid parameters. There was a consensus that a flag for saving intermediate results per plugin would be useful. It is up to the workflow execution platform to support the flag and trigger saving intermediate results per plugin.

When chaining plugins accessing data from file systems, compatibility must be ensured via consensus on supported file formats.

Monitoring information is important as collecting metrics on passed runs is a good approximation of execution profile. More in-depth discussion is needed to define the format and content of an execution profile that could be analyzed programmatically. It is conceivable that each plugin would contain a test inside of a container that could provide an approximation of its execution profile. Since there are no standardized execution profile and error log formats, the workshop consensus was to recommend leveraging existing widely-available tools (e.g., Kubernetes and Kibana + ElasticSearch [5], [6]

The streaming style communication and short vs long lived container execution scenarios need to be researched. More details are provided in Appendix D.

Standard packaging of Web UI modules

Review of existing solutions

Currently, there are no conventions for creating new web user interface (UI) modules that collect information needed by algorithmic plugin containers. The workshop speakers and attendees create UI modules that leverage different libraries (e.g., AngularJS, Angular6, Django, etc.). There is an overlap in collecting information about input image collection and parameters, as well as about output scalar values, vectors, image collections, and prediction models.

Common characteristics of existing solutions and desired best practices

The workshop consensus was to recommend building UI modules that would support data types found on CPU and GPU hardware platforms. Due to the fact that UI modules can include code executed on clients, security certificates for UI modules accompanying a plugin container must be considered.

While a consensus is possible to create reusable UI modules for basic data types (string, int, etc.), the problem of defining UI modules for arrays needs more research.

The workshop consensus was to recommend building UI modules with data types that would be handled (interpreted and rendered) consistently by multiple plugin execution platforms and would enable specifying a valid range of each parameter. "Fancy" custom-made datatypes used in UI modules will have very limited support, at least initially.

Development of UI modules and examples accessible via GitHub repositories would lead the community toward reusability of UI modules and consistent interpretation and rendering of UI by all workflow execution platforms. Detailed discussion notes are provided in Appendix E.

Security of container-based distribution

Review of existing solutions

The workshop consensus was to recommend following best practices to secure containers according to the "Application Container Security Guide" [8]. The contributors of containerized plugins must avoid the main risks, such as (1) compromising the content of an image or container and (2) misusing a container to attack other containers, the host OS, other hosts, etc. The review of [8] included mitigation strategies, such as:

- Tailor the organization's operational culture and technical processes to support the new way of developing, running, and supporting applications made possible by containers
- Use container-specific host OSs instead of general-purpose ones to reduce attack surfaces
- Only group containers with the same purpose, sensitivity, and threat posture on a single host OS kernel to allow for additional defense in depth

- Adopt container-specific vulnerability management tools and processes for images to prevent compromises
- Consider using hardware-based countermeasures to provide a basis for trusted computing
- Use container-aware runtime defense tools.

Since the workshop attendees have been applying mostly home-grown containerized algorithms to big image data, they did not have many inputs about security of downloaded containers from the internet or experiences with the aforementioned risks of using containers.

Common characteristics of existing solutions and desired best practices

For open source algorithmic plugins, we recommend creating registries for Docker images, uploading all code to GitHub for informal reviews, having mechanisms for validating plugins, and restricting write access to registries for Docker images.

More research and in-depth discussions are needed to understand how to design security scanning tools for Docker images and Docker containers, and how to combine the tools in continuous integration/continuous delivery (CI/CD) workflow [12]. Commercial vendors are leading this effort to meet Security Technical Implementation Guides (STIGs) [13] and those efforts can be leveraged in scientific workflows.

Security of workflow execution consisting of chained Docker containers is the responsibility of users selecting containerized algorithmic plugins and system administrators managing the software and hardware resources running the workflow. A consensus is possible to address this security aspect by plugin authors signing a Docker image representing his/her algorithmic plugin. Detailed discussion notes are provided in Appendix F.

4. Summary

This workshop focused on interoperability issues of chained containerized algorithmic plugins running in cluster/cloud environments. The interoperability issues represent specific technical challenges in the abstract reference architecture established under the NIST Big Data Interoperability Framework [14], Volume 6, Fig. 3. In the context of data providers, data consumers, big data application providers, big data framework providers, and system orchestrators, this workshop addressed containerized software tools and algorithm transfer among these big data actors.

The main workshop conclusions are presented in Table 1 below. Table 1 columns correspond to workshop recommendations, possible consensus areas, and future research topics. The rows of Table 1 correspond to discussions focused on:

1. Containerization of execution code
2. Data storage
3. Interoperability requirements of workflow engines for running containerized plugins
4. Standard packaging of web user interface modules
5. Security of container-based distribution.

In summary, the workshop was an initial step toward building a community consensus on creating inter-operable containerized computational plugins. The benefit of such plugins for the microscopy community lies in reusability of algorithmic tools that can be chained into scientific workflows/pipelines and executed in cluster/cloud environments. This written report summarizes the existing solutions, desired best practices, and the research and development goals towards (1) prototype plugins following a community consensus and (2) prototype registries for searchable interoperable plugins. Initial code templates for building containerized algorithmic plugins are available from GitHub[3].

[3] https://github.com/usnistgov/WIPP-Plugins-base-templates

Table 1: Workshop conclusions ranging from practical recommendations to achieved consensus on development and future research and discussions

"Practice:" Workshop Recommendation	**"Development:" Consensus Possible For**	**"Research:" More Discussion Needed to**
Use the on-line best practices for containerization of software [1], [2].	Launching algorithms with parameters using error handling for the "run and then destroy" container scenario.	Construct complex user interfaces for collecting parameters passed to container executions.
Adopt specifications for the metadata manifest files from parallel efforts, e.g. [3], [4] .	Application programming interface (API) to access a spectrum of file formats.	Specify the access API for file formats and capture computational provenance, e.g., [7].
Store plugin manifest metadata files in GitHub and leverage existing tools for storing execution profiles and errors e.g., [5], [6].	The information in plugin manifests to include parameters and execution requirements using a predefined schema.	Define the methods that would assist in finding plugins and analyze execution log profiles.
Build UI modules that would consistently support data types found on CPU and GPU hardware platforms.	Creation of reusable UI modules for basic data types (string, int, …).	Create reusable UI modules for complex data types (arrays, …).
Follow the NIST report entitled "Application Container Security Guide" [1].	Security of plugin execution being addressed by plugin authors signing a Docker image.	Understand how to design container security scanning tools, and how to integrate them CI/CD workflows.

References

[1] "Docker best practices," Docker best practices. [Online]. Available: https://docs.docker.com/develop/develop-images/dockerfile_best-practices/. [Accessed: 10-Jan-2020].

[2] A. Chandrasekaran, "Best Practices for Running Containers and Kubernetes in Production." Gartner, 25-Feb-2019.

[3] OME Team, "Next Generation File Formats for BioImaging," Community Partners, Nov-2019. [Online]. Available: https://forum.image.sc/t/next-generation-file-formats-for-bioimaging/31361.

[4] Amstutz, Peter; Chilton, John, et al., "Common Workflow Language," Common Workflow Language, 02-Feb-2020. [Online]. Available: https://www.commonwl.org/.

[5] "Kibana." [Online]. Available: https://www.elastic.co/products/kibana. [Accessed: 10-Jan-2020].

[6] "Elastic Search." [Online]. Available: https://www.elastic.co/. [Accessed: 10-Jan-2020].

[7] "N5 file format," 02-Feb-2020. [Online]. Available: https://github.com/saalfeldlab/n5.

[8] Murugiah Souppaya, John Morello, Karen Scarfone, "Application Container Security Guide," NIST Special Publication," NIST IR, vol. 800–190, Sep. 2017.

[9] "Hierarchical Definition File Format," HDF5. [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/. [Accessed: 02-Feb-2020].

[10] "Docker - Use multistage builds." [Online]. Available: https://docs.docker.com/develop/develop-images/multistage-build/. [Accessed: 10-Jan-2020].

[11] "2018 Docker Usage Report." [Online]. Available: https://sysdig.com/blog/2018-docker-usage-report/. [Accessed: 10-Jan-2020].

[12] Udi Nachmany and Hayley Denbraver, "Building security into your Azure DevOps Pipeline," 16-Dec-2019. [Online]. Available: https://snyk.io/blog/building-security-into-your-azure-devops-pipeline/.

[13] "DoD Security Requirements Guides (SRGs) and DoD Security Technical Implementation Guides (STIGs)." [Online]. Available: https://public.cyber.mil/stigs/compilations/. [Accessed: 10-Jan-2020].

[14] "NIST Big Data Interoperability Framework (NBDIF)." [Online]. Available: https://bigdatawg.nist.gov/V3_output_docs.php. [Accessed: 02-Feb-2020].

[15] "LXC is a userspace interface for the Linux kernel containment features." [Online]. Available: https://linuxcontainers.org/lxc/introduction/. [Accessed: 10-Jan-2020].

Acknowledgments

The workshop organizers gratefully acknowledge the sponsorship contribution by PerkinElmer.

The authors wish to thank their colleagues who have reviewed drafts of this document and contributed to its technical content during its development, In particular, Mylene Simon (NIST), Joe Chalfoun (NIST), Konstantin Taletskiy (NCATS), Michael Majurski (NIST), Mohamed Ouladi (NIST), Nicholas Schaub (NCATS), Antoine Gerardin (NIST), Sunny Yu (NCATS), Samia Benjida (NIST), Guillaume Sousa Amaral (NIST), Keats Kirsch (NCATS), Derek Juba (NIST), and Philippe Dessauw (NIST). In addition, we would like to acknowledge the inputs from the NIST colleagues, Alden Dima, Zachary Traut, John Henry Scott, and Mary Brady whose comments made the document more readable and complete.

The authors also acknowledge the attendees from the following organizations who participated in the workshop including Allen Institute for Cell Science, Axle Informatics, Chan Zuckerberg Initiative, Howard Hughes Medical Institute (Janelia Research Farm), Medical Science and Computing contracting to BCBB/NIAID/NIH, NCATS NIH, NCI NIH, NIST, NSF, CDRH FDA, National Institute of Technology and Metrology in Paraguay (INTN Instituto Nacional de Tecnología, Normalización y Metrología), National Center for Supercomputing Applications - University of Illinois at Urbana-Champaign, Perkin Elmer, University of California - Santa Barbara, Indiana University, University of Cardiff in Wales, KTH Royal Institute of Technology in Sweden, University of Dundee in Scotland, and Carl Zeiss Microscopy.

Glossary of Technical Terms

Alpine Linux is a Linux distribution built around musl libc (a standard library for Linux-based devices) and BusyBox (a library combining many UNIX utilities). This Linux image is only 5 MB in size and therefore suitable for applications in which containers have to be transferred to remote compute nodes. See https://hub.docker.com/_/alpine/

Android Package (APK) is the package file format used by the Android operating system. It is suitable for distribution and installation of mobile applications. See URL: https://en.m.wikipedia.org/wiki/Android_application_package

AngularJS (or Angular 1) is a JavaScript framework for dynamic web applications that extends HTML's syntax and injects data bindings and application dependencies. See https://docs.angularjs.org/guide/introduction

Angular v2 and above is a TypeScript (a superset of JavaScript) framework for building dynamic web applications. Angular provides many built-in features for easier programming. See https://www.tutorialspoint.com/angular6/index.htm

Apache Arrow is a development platform for handling data in-memory. It consists of libraries for interprocess communication and zero-copy streaming, and it supports development in 10 different programming languages. See https://arrow.apache.org/

Application container technologies (or software containers) are a form of operating system virtualization that are used for packaging software application packaging. Containers simplify managing application dependencies and software executions in distributed computational environments. See [8].

A Docker container image is a software container that includes code, runtime, system tools, system libraries and settings. Docker containers can be executed using a Docker engine. See https://www.docker.com/resources/what-container

Containerized algorithmic plugin for cluster/cloud execution is the software container and its manifest file. The software container can be executed from a command line with a set of arguments. The arguments and hardware requirements are described in a manifest file. Inter-operable plugins can be chained into a computational workflow/pipeline.

Kibana is a visualization of cloud-based execution information. It visualizes performance metrics and log files from Elasticsearch input data. Elasticsearch is an analytics engine designed for monitoring computational infrastructure and security analytics. See https://www.elastic.co/products/kibana

Kubernetes (K8s) is a management system for automating deployment and execution of containerized applications. It has many features including horizontal scaling of container-based executions and storage orchestration which are important for massive data analyses. See https://kubernetes.io/

N5 file format is focused on the application programing interface (API) that is targeting storage of large chunked n-dimensional tensors, and arbitrary meta-data in a hierarchy of groups similar to HDF5 [9]. See [7].

Slurm workload manager is a computational job scheduling system for Linux clusters. It allocates compute nodes, starts and monitors the execution, and manages a queue of pending jobs. See https://slurm.schedmd.com/overview.html

Snyk is one of the tools in the Azure suite of tools that is designed for container security scanning (Snyk Security Scan Azure Pipelines Task). It is integrated into continuous integration, continuous development (CI/CD) pipelines in Azure. See https://snyk.io/blog/building-security-into-your-azure-devops-pipeline/

Zarr file format is focused on providing support for chunked, compressed, N-dimensional arrays. It depends on NumPy and other Python libraries. See https://zarr.readthedocs.io/en/stable/

Appendix A: Pointers to Image Analysis Projects Mentioned During Day 1

- Allen Cell Explorer - The Allen Cell Explorer is the data portal for the Allen Institute for Cell Science, where you can explore publicly available data, tools and models.
  - URL: https://www.allencell.org/about.html
- BisQue - is a free, open source web-based platform for the exchange and exploration of large, complex datasets.
  - URL: https://bioimage.ucsb.edu/bisque
- BrownDog - Brown Dog's goal is to prototype a highly distributed and extensible science driven Data Transformation Service (DTS)
  - URL: https://browndog.ncsa.illinois.edu/
- Chan-Zuckerberg Initiative projects - Repositories for Essential Open Source Software for Science.
  - URL: https://github.com/chanzuckerberg
- Clowder - Open Source Data Management for Long Tail Data
  - URL: https://clowder.ncsa.illinois.edu/
- IDR - The Image Data Resource (IDR) is a public repository of image datasets from published scientific studies, where the community can submit, search and access high-quality bio-image data.
  - URL: https://idr.openmicroscopy.org/
- ImageJ/Fiji and related projects - Fiji is an image processing package—a "batteries-included" distribution of ImageJ, bundling plugins which facilitate scientific image analysis.
  - URL: https://en.wikipedia.org/wiki/ImageJ
- ImJoy - is a plugin powered hybrid computing platform for deploying deep learning applications such as advanced image analysis tools.
  - URL: https://imjoy.io
- LIMPID - NSF funded project focused on a large-scale distributed image-processing infrastructure
  - URL: https://www.nsf.gov/awardsearch/showAward?AWD_ID=1664172
- OMERO - is client-server software for managing, visualizing and analyzing microscopy images and associated metadata.
  - URL: https://www.openmicroscopy.org/
- Python-based image processing - refers to scikit-image library that is a collection of algorithms for image processing.
  - URL: https://scikit-image.org/
- PerkinElmer Image Analysis – The PreciScan optional plug-in for Harmony high-content analysis software uses intelligent image acquisition to enable a fully automated workflow of low magnification pre-scan, image analysis and higher magnification re-scan
  - URL: https://www.perkinelmer.com/product/preciscan-instrument-license-hh17000003
- Scientific Computing Software at Janelia - provides full software life cycle support for Janelia's project teams, labs and shared resources.
  - URL: https://www.janelia.org/support-team/scientific-computing-software
- WIPP - Web Image Processing Pipeline (WIPP) has been designed for enabling interactive measurements and discoveries over very large images
  - URL: https://isg.nist.gov/deepzoomweb/software/wipp
- Zeiss APEER - Cloud-based Digital Image Processing Platform
  - URL: https://www.apeer.com/home/

Appendix B: Working Group 1 - Containerization of execution code

Moderator: Mylene Simon (NIST)
Scribe: Joe Chalfoun (NIST)

Best practices for how to build code containers for plugins. This includes both the order of steps and the steps themselves.

Session 1: Review of existing solutions

This session was driven by a set of questions about containers being used in designing computational plugins for cloud-based execution.

- Are you using containers? Which technology/engine? [7], [8]
  - Docker, CoreOS Rocket (rkt), Apache Mesos Containerizer, Linux Containers (LXC), Singularity, other. – see references
- What are you containerizing?
  - Individual algorithms, Complex programs, Web applications, other.
  - Which programming languages? All and any
  - Does your container expose any UI?
- Is your container accessing external data? How?
  - Volume/folder mounts, database access, download, other.
- Are you disseminating your containers? Where?
  - Public repositories (DockerHub, other)
  - Private repositories
- Are you using any container orchestration technology? Which one(s)?
  - Kubernetes, Docker Swarm, Apache Mesos, other.

The underlined words correspond to the most frequently used solutions.

Session 2: Common characteristics

This session collected information about tools, preferences and challenges in building containers.

- Best practices for writing Dockerfiles
  - Summary: See the best practices URL in the references
- Reducing size of Docker images
  - Discussion: Tools for packaging: Ubuntu, Conda (reduce the final size)
  - Discussion Preferences: multi-stage builds, Jenkins continuous integration, package a small scope into each container, design an automatic installation launching system to download the dependencies needed to run a container
  - Discussion Challenges of automated launching system: reproducibility of results vs upgrading the code in containers
  - Additional topics: Granularity of plugins - what should be packaged into a container in terms of functionality? In terms of container size? In terms of the number of dependencies?
  - Summary: Use Alpine images as base images, multi-stage builds, and continuous integration tools
- Container entrypoints
  - ENTRYPOINT execution vs shell form, command line interface (CLI)
  - Standardization of CLI and parameters handling (keyword arguments/options, positional arguments, …)
  - Best practices for default help menu and parameters sanity checks

    - Discussion: Many similarities in between existing Zeiss APEER and WIPP systems
    - Summary: a consensus is possible
- Data access
    - Mounting folders/volumes
    - Standards for mount paths? /data/inputs and /data/outputs for example
    - File permissions, define USER in Dockerfile, use "—user" to run as specific user, etc.
    - Error handling when data cannot be found or cannot be accessed
    - Summary: a consensus is possible
- User Interface (UI)
    - EXPOSE ports for web UI
    - More problematic with Graphical UI (GUI)
    - Summary: More in-depth discussions are needed
- Error handling
    - Returning exit codes
    - Message logging
    - Discussion: There are three scenarios for running a container:
        - Run and then destroy
        - Run forever until you close the browser
        - Run, pause, and restart from an intermediate state
    - Discussion: The majority of implementations support only "run and then destroy" scenario.
    - Discussion: We need a consistent value for an error value returned by each program (i.e., 0 – success, 1 – fail) which is not the case in all programming languages. All workflow platforms can check the status of each container output and stop the executions of plugin containers in the chain.
    - Summary: an initial consensus about error handling can be achieved for the "run and then destroy" scenario.

Additional discussion:

- Dissemination of Docker images
    - Tagging images – conventions
    - Public Docker registry (DockerHub)
    - Private Docker registries
    - Discussion: Most of the institutions will host their own private repository and implement their own tagging system before moving Docker images into public repositories.
    - Summary: a registry of Docker images might be useful to search and find functionalities needed for creating complex image analysis pipelines executed in cloud-based environments
- Orchestration technologies
    - Kubernetes, Docker Swarm, Apache Mesos, other.
    - Other container solutions
    - CoreOS Rocket (rkt) by CoreOS/RedHat
    - Mesos Containerizer by Apache
    - Linux Containers LXC
    - Singularity
    - Summary: While Kubernetes is the most frequently used container execution orchestration middleware among the workshop attendees, the computational plugins can become inter-operable in platforms based on other orchestration technologies (assessment as of the workshop date)
- Performance
    - What is the tradeoff of Docker vs native execution?
    - Discussion: Conda containerization is preferred instead of Docker because Docker doesn't work well on Windows.

- Summary: Discussions with vendors might be beneficial

Appendix C: Working Group 2 - Data storage and access interfaces

Moderator: Nathan Hotaling (NCATS NIH)
Scribe: Michael Majurski (NIST)

How the inputs/outputs of plugins should be formatted for tabular data and image data. Also, how metadata will be handled between plugins.

Sessions 1 and 2: Review of existing solutions and common characteristics

This session was driven by a set of questions about accessing data from computational plugins for cloud-based execution.

- Should we standardize how to access data from computational plugins? (Interfaces: databases, file systems).
  - What type of data? binary, tabular, images, metadata, annotations?
    - Discussion: For both databases and file systems/object store, if you need something specific it is on you to write the driver. The system should not have to support every bespoke database.
  - What about institutional databases? Where a set of plugins needs to call and get data from a shared data resource managed by a specific institution.
    - Discussion: You have fewer people developing systems than there are people developing plugins. Therefore, you want to push the effort of dealing with bespoke formats to the plugin developers.
  - What about provenance tracking?
    - Discussion: For both databases and file systems/object store, you need to handle writing data provenance data and enable two-way transfer to push the results/provenance data back into the system. For example, if you have tools which are not inside the system: how do you get its results back into the system?
  - Discussion opinions:
    - Large binary images make more sense to write to file system. Certain data types make more sense to write into databases.
    - You cannot solve all storage and access problems with one system. For example, large tabular data, you need Cassandra, with large 10 TB image data you need specific file system tools to handle. However, we should target the largest number of users, not all use cases.
    - Need to standardize how data access from plugins is required in order to support inter-operable plugins.
    - While you can translate between different formats, it is a slow process, and it requires constant developer time which is not something we should be doing. It is practically difficult to specify this from a funding and justification perspective.
    - Does the container orchestration standard specify what this mount looks like and can be? What is happening inside the Docker container is different from what is happening outside, what you mount to the container.
  - Summary: The problem of designing interfaces to databases and file systems to enable consistent programmatic access to data from computational plugins requires additional in-depth discussions. The discussion opinions varied. While there is a need to build standard interfaces in order to expand the current limited support of mounting folders/volumes on a file system, there is also concern that a standard interface would lock developers into a technology window. In-depth discussions are needed on how to incorporate access to databases (and filesystems) that are continually being developed. The provenance tracking should be included in those discussions as well.

    - Note: In case where a database should be supported, if so which database? This question became a moot point after discussing question 1.

- What file formats should be universally supported by computational plugins for loading and for writing to a file system?
    - Should we select a set of file formats?
    - Discussion inputs:
        - Select a subset of file formats based on mime types
        - Suggested candidate file formats - Apache Arrow, Zarr, N5, tiled tiff, BigDataViewer (BDV) format.
    - Discussion opinions:
        - Currently the most flexible format is something like Zarr or N5, however that places significant burden on the plugin developers to support that specific format. The reason csv is appealing is that all languages support reading it. It is tempting to go with a modern format, but that might exclude certain members of the programming world.
        - There is a distinction between what the system accepts, vs what is permissible. If you expect people to chain plugins, developers must know what to write and how to interpret the input coming from the previous plugin.
        - Plugins must very clearly define the input and output types (image, vector field, point cloud) in a way that every plugin agrees with to enable plugins to chain together. If the next plugin needs to load a point-cloud, there cannot be 3 or 4 point-cloud formats that can be loaded without specifying which one the input is.
        - Should the system auto convert the file into a tabular array in memory format like Apache Arrow?
    - Discussion opinions:
        - Pick top 5 or 7 languages where you write a middle layer (in each language) enabling lots of in memory communication between languages used in mini-plugins/files
        - Example: python files calling each other, passing around numpy like arrays which can then be automatically wrapped up into a chain of python files with the self-written converter into a single web image plugin docker container where all intermediate steps are kept in memory, within that chain of python code.
    - Summary: a consensus is possible to specify an access API to a spectrum of file formats. Among the newer file formats suitable to big image analyses, Apache Arrow, Zarr, N5, and tiled Tiff formats have been mentioned frequently as good candidates for specifying the initial implementations of the access API. More in-depth discussions are needed to specify the access API for file formats.

- How to handle metadata describing inputs and outputs of computational plugins?
    - Do we need an ontology of input and output types? Where should that metadata be stored? Should the storage options include a database and a file system?
    - Discussion inputs:
        - Need some standard for plugin metadata in order to create inter-operable plugins.
        - The file format can be JSON or it can be XML embedded in the image.
    - How do we specify and store the plugin metadata to enable chaining?
    - Discussion opinions:
        - While specifying the file format for metadata is easier, what to include in the information stored in that metadata file is a harder question.
        - JSON file format for metadata (in filesystem or filestore), N5 or Zarr for specific files on filesystem since they are very flexible formats.
        - What about we write converts to some basic formats, tiled tiff, csv; where we write converters that might be slow, but you can use the system if you get your file into csv. Writing a converter

to something old which enables lots of backward compatibility, while still pushing the field toward a modern file format.
        - What about defining a middle layer for API access (like N5) which can be backed by many different modern big data block-based storage systems?
        - More modern access APIs should be laid out so there is better separation between the underlying file formats (most microscope vendors wrap headers onto a tiff and getting to the underlying tiff is difficult) with better differentiation and separation it might be able to better disentangle the underlying data and any metadata that the vendor has added on top of the fundamental data type.
        - How to avoid a race to the lowest common denominator (pixel size, shape, channels) which provides enough metadata for a large majority of the applications, but will not support all use cases?
    - Discussion inputs:
        - There are large national level efforts in this space (13 at NIH, 1 at NIST). There are efforts in Japan to do the same thing. They are trying to define what a "Minimal Metadata Specification" should be. Examples: Bio-formats, Open Microscopy Environment (OME)
        - Keep apprised of the national efforts in order to decide what to store in metadata files.
        - Example: next gen metadata standard software development effort – see https://forum.image.sc/t/next-generation-file-formats-for-bioimaging/31361
    - Discussion opinions:
        - Don't wait for a global standard, do some work around a specific topic which might be incorporated into a larger standard later.
        - Metadata codebases exist that should be vetted and tested
        - There is a big change that is happening; the OME tif model has run out of steam because the world is changing around it as the applications have shifted.
        - How the metadata is stored is less of a concern than the fact that I can access that metadata. So if there are people who do care about how the metadata should be stored, people will follow those leaders, since they will think deeper about the metadata challenges than the uses who just want the metadata access.
    - Summary: In order to enable chaining of computational plugins, input and output file formats should become parts of the metadata stored following JSON or XML metadata formats. More in-depth discussions are needed to specify the metadata fields describing input and output file formats in each plugin.

Additional questions that were not discussed due to time limits.

- Should we allow plugins to create/store a database or any random file type? Or should we prescribe it somehow?
- Should containers define data access requirements (e.g. data load time)?
- What are requirements for sensitive data?
- What concerns are there for filesystems that we should include or not include in plugins?
- Quirky file systems not supported?
- What about HPC/singularity systems vs Kubernetes?

Appendix D: Working Group 3 - Interoperability requirements

Moderator: Peter Bajcsy (NIST)
Scribe: Nicholas Schaub (NCATS NIH)

Plugins are chained together to form a workflow. However, how to execute this workflow is not standardized nor is the format for how to describe this workflow.

Session 1: Review of existing solutions

The cumulative answers to specific questions about existing solutions are provided below in the tables. By summarizing all table entries, the most frequent answers are:

- Workflow editors: Argo and custom solutions,
- Job schedulers: K8 and SLURM,
- Workflow orchestration: Kubernetes,
- Workflow representation: scripting files (.py, .js) and Argo file (.yml),
- Workflow monitoring: Kibana + ElasticSearch or none,
- Supported hardware: CPU and GPU platforms,
- Passing Parameters from Clients to Servers: JSON files and a list of arguments,
- Finding and Fetching containerized steps of workflows: private and public registries,
- Finding and accessing image collections: native file systems,
- Logging errors and warnings: language specific logging and none,
- Target hardware: Commercial Cloud, cluster on premises, and multiple CPU/GPU machines,
- Size/Volume of Data Expected to Process: GB, TB, and PB,
- Is DAG Sufficient? Yes

Table 2: Workflow Editors

Tool	Number of Responses
Argo	6
Custom/In house	3
KNIME	1
Airflow	1

Table 3: Job Schedulers

Tool	Number of Responses
K8s/Argo	3
None/Not Applicable	3
Slurm	3
Hedgehog	1
Custom/In House	1

SGE	1

Table 4: Workflow Orchestration

Tool	Number of Responses
Kubernetes	9
Terraform	2
Snakemake/KNIME	1
Docker Swarm	1
None/Not Applicable	1

Table 5: Workflow Representation

Format	Number of Responses
Argo yml	6
Scripting files (.py, .js)	2
Custom/In House	1
Not Applicable	1

Table 6: Workflow Monitoring

Tool	Number of Responses
Kibana + Elasticsearch	4
Not applicable/None	3
ECK	1
Prometheus + Grafana	1
Airflow UI	1
Argo Dashboard	1
Custom/In House	1

Table 7: Supported Hardware

Platform	Number of Responses
CPU	8
GPU	8
CPU&GPU cluster	3

Virtual Machines (VMs)	3
HPC	2
Cloud	1
Elastic Compute	1

Table 8: Passing Parameters from Clients to Servers

Format	**Number of Responses**
JSON	7
List of arguments	2
Environment Variables	1
Airflow REST API	1
Yml	1
Nd array	1

Table 9: Finding and Fetching containerized steps of workflows

Containers	**Number of Responses**
Public registries	8
Private registries	2
No containers	1
Custom	1
None	1

Table 10: Finding and accessing image collections

Access mechanism	**Number of responses**
Native File System	6
REST	1
Database	1

Table 11: Logging errors and warnings

Tool	Number of responses
None	2
Language specific logging	2
Airflow logs	1

Table 12: Target hardware

Hardware	Number of Responses
Commercial Cloud	6
Cluster on premises	5
Multiple CPU/GPU	4
Personal computer	1
HPC	1

Table 13: Size/Volume of Data Expected to Process

Size	Number of Responses
TB Scale	7
GB Scale	6
PB Scale	3

Table 14: Is DAG Sufficient?

Response	Number of Responses
Yes	6
No. Support dynamic workflows	1
Probably/Maybe	1

Session 2: Common characteristics

This session was driven by a set of questions about interoperability requirements of workflow engines for running containerized plugins:

- Finding and fetching containerized steps/plugins to form workflows (Access? Curation? Search? Retrieval?)
  - Discussion opinions:
    - GitHub could become the repository of plugin manifest metafiles

        - Since we are handling binary files as well, we would need to encode package management environments, such as, npm, conda, docker hub, into plugin manifest files
        - How would one filter through those plugins in GitHub repositories? Can we add tags and use Google text search?
            - We need to standardize a taxonomy/ontology for plugin tags
            - We can create highly structured fields to assist filtering for plugins
        - Visualization using Common Workflow Language (CWL) Viewer (https://view.commonwl.org/) if we follow CWL
        - (super) users will need scripting to create automated search
    - Summary: As a starting point, plugin manifest metafiles are recommended to be stored in GitHub and labeled with a tag (i.e., encoded in a file name). More in-depth discussion is needed to define the plugin tags and the methods that would assist in finding plugins.

- Workflow configuration using visual programming to specify links, hardware platforms, and scheduling/orchestration mechanisms (Should there be information embedded in plugins to support workflow configurations?)
    - Discussion opinions:
        - While DAG representation is probably enough to capture complexity of workflows, we would need support for a streaming style communication between plugins during workflow execution
        - We would need to add to plugin manifests whether a plugin container is short lived or long lived
        - Note: in reference to three scenarios for running a container:
            - Run and then destroy
            - Run forever until you close the browser
            - Run, pause, and restart from an intermediate state
    - Summary: There was a consensus that the information in plugin descriptors/manifests should include supported I/O file formats and hardware requirements using a predefined schema (e.g., JSON or XML). The streaming style communication and short vs long lived container execution scenarios need to be researched.

- Workflow representations for DAG (Is there any information that should be stored in plugin descriptors to facilitate I/O compatibility of chained plugins?)
    - Discussion inputs:
        - It is necessary to pursue a compatibility of inputs / outputs
    - Summary: When chaining plugins accessing data from file systems, a compatibility can be achieved via consensus on supported file formats.

- Passing parameters from clients to servers in order to parametrize workflow DAGs (can we check that a plugin in a WF has all parameters within a range? Should each parameter come with a range definition?)
    - Discussion inputs:
        - For plugin execution, it is needed to define a parameters range for inputs / outputs
        - Note: some info will only be accessible at runtime
    - Summary: The plugin descriptors should include not only a list of parameters but also the fields defining the range of valid parameters.

- Passing data from storage to execution locations according to parametrized workflow DAGs (Should we include the flag in for saving intermediate results per plugin? Access efficiency?)
    - Summary: There was a consensus that the flag would be useful. It is up to the workflow execution platform to support the flag and trigger saving intermediate results per plugin

- Workflow monitoring for purpose

- Performance Stats: utilization of RAM & CPU per plugin, execution time per plugin
- Debug information: error and warning messages per plugin (common notation for auto parsing?
- Summary: Monitoring information is important as collecting metrics on passed runs is a good approximation of execution profile. More in-depth discussion is needed to define the format and content of an execution profile that could be analyzed programmatically. It is conceivable that each plugin would contain a test inside of a container that could provide an approximation of its execution profile. Since there are no standardized execution profile and error log formats, the workshop consensus was to recommend leveraging existing widely available tools (e.g., Kubernetes and Kibana+ElasticSearch).

Appendix E: Working Group 4 - Standard packaging of web UI modules

Moderator: Sunny Yu (NCATS NIH)
Scribe: Samia Benjida (NIST)

Plugins need a GUI to configure their settings and outputs appropriately for the user. Can we agree on a schema, format, and scope of what will be universally supported for all UIs?

Sessions 1 and 2: Review of existing solutions and common characteristics

This working group combined the two sessions by posing and discussing the following questions during the allocated time:

- What type of data to support?
  - Discussion Inputs:
    - Basic types must be supported (string, int …). The case of arrays is more complicated since it can represent an image, a polygon, or annotations.
    - We must take into account that the shape of the data does not define the data.
  - Discussion about open questions:
    - How to define arrays?
    - How to define multidimensional data?
    - Should we implement an Android Services Library (android package or APK file like a library)?
  - Summary: While a consensus is possible to create reusable UI modules for basic data types (string, int, ...), the problem of defining UI modules for arrays needs more research.

- How do we provide flexibility and standardization in data types?
  - Discussion Inputs:
    - We need to support both GPU and CPU
    - It is not necessary to have a security certificate to privatize the plugins.
  - Summary: The consensus of the workshop is to recommend building UI modules that would support data types found on CPU and GPU hardware platforms. Due to the fact that UI modules can include code executed on clients, security certificates for UI modules accompanying a plugin container are an option.

- What is standard for how to create new UI modules?
  - Discussion Inputs:
    - A datatype that corresponds to a UI module for an algorithmic plugin can range from a simple layout to custom-made. The custom-made datatypes should be used only for optional parameters and should not be handled by the developer of the platform.
    - It should be possible to specify parameter ranges of values.
    - There is still a question about consistency of interpreting and rendering UI modules on different plugin execution platforms.
  - Summary: The workshop consensus was to recommend user build UI modules with datatypes that would be handled (interpreted and rendered) consistently by multiple plugin execution platforms and would enable specifying a valid range of each parameter.

- How do we support extensibility in the user interface?
  - Summary: “Fancy” custom-made datatypes used in UI modules will have initially very limited support.

- How should we handle the migration to the new solution?
  - Discussion Inputs:
    - The future plugins developed by the interested companies should be compatible.
    - A GitHub repository should be set up so that people can check how the standards are being implemented and adapt their plugins.
  - Summary: Since there has been very little work in standardizing reusable UI modules for algorithmic plugin containers, examples in GitHub repositories would lead the community toward reusability of UI modules and consistent interpretation and rendering of UI by all workflow execution platforms.

Appendix F: Working Group 5 - Security of container-based distribution

Moderator: Keats Kirsch (NCATS NIH)
Scribe: Derek Juba (NIST)

How do we monitor, enforce, and encourage good security practices for plugins and their executions?

Session 1: Review of existing solutions

Review NIST report as a container security reference:
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

Risks

- Compromise of an image or container.
- Misuse of a container to attack other containers, the host OS, other hosts, etc.
- Other?

Mitigation Strategies

- Tailor the organization's operational culture and technical processes to support the new way of developing, running, and supporting applications made possible by containers.
- Use container-specific host OSs instead of general-purpose ones to reduce attack surfaces.
- Only group containers with the same purpose, sensitivity, and threat posture on a single host OS kernel to allow for additional defense in depth.
- Adopt container-specific vulnerability management tools and processes for images to prevent compromises.
- Consider using hardware-based countermeasures to provide a basis for trusted computing.
- Use container-aware runtime defense tools.

Session 2: Common characteristics

This session evolved into discussing the following questions during the allocated time:

1. How to address security of container-based distribution via registries for Docker images representing algorithmic plugins?
2. How hard is it to scan containerized algorithmic plugins? What tools exist?
3. Who should be responsible for security of Docker container execution? Plugin author? Plugin repository manager? User? How to deliver secure plugins?

1. Security of registries for Docker images representing algorithmic plugins
   - Discussion inputs:
     - Nobody is signing plugin containers right now
     - GitHub can be turned into a plugin registry by adding JSON file (plugin manifest file). Plugin manifest files can be downloaded from different GitHub repositories. If source code packaged into plugin Docker images is hosted in GitHub, then informal review can be done, and Docker images can be built by the users. There is a need for plugin validation (i.e., tests) and for testing security of Docker containers.
     - Security can be provided through a list of authorized users with varying permissions to run algorithmic plugins.

- Summary: For open source algorithmic plugins, registries for Docker images can become "trusted" by putting all code to GitHub for informal reviews, having mechanisms for validating plugins, and by restricting access to registries for Docker images.

2. How hard is it to scan plugins? What tools exist?
   - Discussion Inputs:
     - Docker has some tools for container scanning.
     - Even non-malicious code can cause problems by unintentionally misbehaving.
     - Services such as "the Snyk Security Scan " could be useful. "The Snyk Security Scan Azure Pipelines Task scans your application dependencies and container images for open source security vulnerabilities."
     - Awareness of security is important. May require a cultural change.
   - Discussion Opinions:
     - Can you trust certifications? Can you identify who certified it?
     - Will security updates break reproducibility?
     - Do plugins have network access? May need to access large external datasets.
     - Open-source containers are more trustworthy. Closed-source containers require trusting the author.
     - Many organizations feel they do not have resources to guarantee security. In this case:
       - Make "best effort".
       - Rely on the threat of punishment to prevent malicious behavior.
   - Summary: More research and in-depth discussions are needed to understand how to design security scanning tools for Docker images and Docker containers, and how to combine the tools in continuous integration/continuous delivery (CI/CD) workflow. Commercial vendors are leading this effort to meet Security Technical Implementation Guides (STIGs) [13] and those efforts can be leveraged in scientific workflows.

3. Who should be responsible for security of Docker container execution? Plugin author? Plugin repository manager? User? How to deliver secure plugins?
   - Discussion Opinions:
     - Users don't want to spend the time and effort to scan, but they are the group who are ultimately responsible for security, since they are the ones who will be impacted by vulnerabilities.
     - Security requirements depend on the identity and resources of those who you expect would want to attack you (your threat model).
     - Plugin author should sanitize inputs.
     - Would ranking/reviews for plugins become an incentive to make plugins secure?
     - Plugin authors should sign containers. Signatures need to be checked.
     - How do you determine what signatures to accept? Manually validate identities by directly contacting the signers?
     - Should communication between plugins be required to be encrypted? HTTPS should generally be required.
     - Security is a lot of work. One should use Docker security features, best-practices, and namespaces (filesystems and privileges) [15]
   - Summary: Security of workflow execution consisting of chained Docker containers is the responsibility of users selecting containerized algorithmic plugins and system administrators managing the software and hardware resources running the workflow. A consensus is possible to address this security aspect by plugin authors signing a Docker image representing his/her algorithmic plugin.