Toward sustainable deployment of distributed services on the cloud: dockerized ODI-PPA on Jetstream

Raymond W. Perigo, Arvind Gopu, Michael D. Young, Yuanzhi Bao

Raymond W. Perigo, Arvind Gopu, Michael D. Young, Yuanzhi Bao, "Toward sustainable deployment of distributed services on the cloud: dockerized ODI-PPA on Jetstream," Proc. SPIE 10707, Software and Cyberinfrastructure for Astronomy V, 107072X (6 July 2018); doi: 10.1117/12.2313647

SPIE.

Event: SPIE Astronomical Telescopes + Instrumentation, 2018, Austin, Texas, United States

Toward sustainable deployment of distributed services on the cloud: dockerized ODI-PPA on Jetstream

Raymond W. Perigo, Arvind Gopu, Michael D. Young, and Yuanzhi Bao

Indiana University, 2709 E 10th St., Bloomington, IN 47408, USA

ABSTRACT

The One Degree Imager - Portal, Pipeline and Archive (ODI-PPA) - a mature & fully developed product - has been a workhorse for astronomers observing on the WIYN ODI. It not only provides access to data stored in a secure archive, it also has a rich search and visualization interface, as well as integrated pipeline capabilities connected with supercomputers at Indiana University in a manner transparent to the user. As part of our ongoing sustainability review process, and given the increasing age of the ODI-PPA codebase, we have considered various approaches to modernization. While industry currently trends toward Node.js based architectures, we concluded that porting an entire legacy PHP and Python-based system like ODI-PPA with its complex and distributed service stack would require too significant an amount of human development/testing/deployment hours. Aging deployment hardware with tight budgets is another issue we identified, a common one especially when deploying complex distributed service stacks. In this paper, we present DockStream (https://jsportal.odi.iu.edu), an elegant solution that addresses both of the aforementioned issues. Using ODI-PPA as a case study, we present a proof of concept solution combining a suite of Docker containers built for each PPA service and a mechanism to acquire cost-free computational and storage resources. The dockerized ODI-PPA services can be deployed on one Docker-enabled host or several depending on the availability of hardware resources and the expected levels of use. In this paper, we describe the process of designing, creating, and deploying such custom containers. The NSF-funded Jetstream led by the Indiana University Pervasive Technology Institute (PTI), provides cloud-based, on-demand computing and data analysis resources, and a pathway to tackle the issue of insufficient hardware refreshment funds. We briefly describe the process to acquiring computational and storage resources on Jetstream, and the use of the Atmosphere web interface to create and maintain virtual machines on Jetstream. Finally, we present a summary of security refinements to a dockerized service stack on the cloud using nginx, custom docker networks, and Linux firewalls that significant decrease the risk of security vulnerabilities and incidents while improving scalability.

Keywords: Keywords: docker, node.js, jetstream, cloud, ODI-PPA, portal, sustainability, nginx

1. INTRODUCTION

The Scalable Compute Archive (SCA) team at Indiana University [1] operates several web portal based software systems that streamline secure data archival and provide access to custom scientific workflows via intuitive graphical user interfaces. SCA portals provide a seamless pathway for scientists in various domains including astronomy, microscopy, radiology, and neuroscience to Indiana University's compute and storage resources. One of the flagship SCA systems, the One Degree Imager - Portal, Pipeline, and Archive (ODI-PPA) [2] is a collection of PHP and Python applications designed to provide astronomers with a single point of access to: 1) data produced by the One Degree Imager (ODI) on the WIYN 3.5m telescope at Kitt Peak, Arizona; 2) integrated science pipelines to process their datasets and; 3) advanced tools to visualize and analyze these data products. It has been and continues to be a reliable workhorse for research groups observing on ODI.

While ODI-PPA's distributed architecture adheres to modern microservice standards to the extent possible, a significant chunk of the codebase uses once state-of-the-art but now legacy frameworks and libraries. This has the potential for increased difficulty in maintaining the service over time. Successive updates to the underlying host operating system carry a risk of breaking dependencies, and increasing the amount of time triaging bugs

Further author information, contact rperigo@iu.edu

Software and Cyberinfrastructure for Astronomy V, edited by Juan C. Guzman, Jorge Ibsen, Proc. of SPIE
Vol. 10707, 107072X · © 2018 SPIE · CCC code: 0277-786X/18/$18 · doi: 10.1117/12.2313647

- time that otherwise could be put toward research and development of newer features or integration of newer science pipelines.

As best-practices for deployment and development of software solutions evolve over time, the SCA team attempts to follow these industry trends, including:

- Node.js [3]- to develop a modern web application including a backend Application Programming Interface (API) all in one programming language (Javascript).
- Docker [4] - to containerize services as well as to deploy commonly used dependencies like database servers and legacy software solutions that do not adhere to current protocols, especially vulnerable web-services with distinct and dated software needs (e.g. a Java application with a specific old and vulnerable version of Oracle Java).
- nginx [5]- to serve as the public and user facing web server for one or more web applications. Additionally to handle load-balancing and high availability needs, should a project require that. In the case of legacy web-applications, this allows us to funnel secured traffic into the appropriate Docker container without connecting it directly to the external network.

As with software, industry trends in how to best utilize the hardware at hand are changing as well. The SCA team operates virtualized server hardware solutions to host projects such as the ODI-PPA. We also use Enterprise-class virtual hardware solutions like the IU Intelligent Infrastructure when budgets allow. However, it behooves us to be prepared, should the need arise to deploy service stacks like the one for ODI-PPA on alternative hardware options on the cloud. Use of solutions like Amazon Web Services and Google Cloud are becoming common place but incur a cost, it was most prudent for us to explore a cost-free alternative like the NSF-funded Jetstream cloud solution.

Given the above, it behooved the SCA team to investigate the proper path forward to sustain ODI-PPA. While porting the complete ODI-PPA stack to an entirely different architecture and programming language is simply too arduous and time-consuming a task given personnel and budgetary constraints, we nevertheless wanted to bring the project in-line with modern best-practices to the extent possible. In this paper we describe such a solution and includes a proof of concept deployment called ODI DockStream - the web portal is accessible to the public at https://jsportal.odi.iu.edu*.

2. TOWARD SUSTAINING SOFTWARE WITH LEGACY COMPONENTS

As mentioned in the previous section, Scalable Compute Archive team has leveraged Linux containers in the form of software from Docker in order to bring order to complex software deployments, as well as provide increased security and reliability.

Docker provides a method for administrators to create containers which house the code for a given application as well as all of its dependencies, without affecting the host operating system or libraries. This allows us to package a suite of applications in such a way that deployment time is greatly reduced without losing the flexibility inherent to the microservice architecture it might use. ODI-PPA containers may still be set up on multiple hosts, should the need arise, but may also be co-located on the same physical or virtual machine.

Additionally, Docker allows us to create arbitrary virtual networks to which we can bind these containers. This provides a method by which sensitive components of the application stack can be completely isolated from the outside world. This helps us to decrease the attack surface when dealing with legacy code.

*The ODI DockStream web portal includes nearly all of the functionality offered by the production ODI-PPA portal (https://portal.odi.iu.edu) but serves a limited subset of data taken between January - October 2016.

2.1 ODI-PPA Service Stack

Currently, the production ODI-PPA service stack is deployed such that each component runs on its own virtual machine in order to avoid interference from other services' dependencies. Having decided to leverage Docker going forward, we had to make decisions on how to best maintain the service's flexibility while streamlining deployment as much as possible. To that end, we eschewed a monolithic approach and have instead developed custom Docker containers for each component of the ODI-PPA stack, utilizing a shared and reusable base container wherever possible. This is also consistent with Docker best practices.

The ODI-PPA service stack is described in detail elsewhere [2]. Below we briefly describe the components that have been packaged into Docker containers:

- The ODI Portal
  - The PHP/Zend application running on Apache, with hooks into a MySQL database, that acts as the frontend for this application stack. The Portal provides researchers with an intuitive interface for ingesting, processing, cataloging, and downloading their data products from the telescope.
  - The base container for this portal provides an Apache installation with a number of PHP modules and the Zend application framework. This is reusable for other PHP-Zend projects under the SCA umbrella, should the need arise.
- The ODI Data Sub System (DSS)
  - A Python application controlling data movement between an archive source and scratch space. This allows for images "frozen" in the archive to be "thawed" and presented to the user for download without consuming excessive amounts of real-time storage. This container can also be used to archive new images from the instrument via a command-line flag at runtime.
- The ODI Worker Node (WN for workflows)
  - A Python application to set up jobs when a user executes a download workflow or one that run image processing pipelines against datasets.
- Additionally, the ODI-PPA stack requires a number of other dependencies in order to function correctly.
  - A MySQL / MariaDB to act as a metadata store archived raw and processed images.
  - A RabbitMQ message broker to ensure all components can communicate with one another regardless of their location.
  - An Nginx server, to act as a gateway to these containers via the Web.

2.2 Docker Container Architecture & Run Scripts

While we have taken great care to package all of the dependencies and application code required for various ODI-PPA components to function, we do not want to have to rebuild the containers every time the service is deployed on a new host, nor do we want our sensitive configuration data (such as usernames and passwords) to be included in the default image. To that end, we have created a set of deployment and activation scripts which provide the needed configuration data to each container at runtime. Additionally, a separate network is created on the host without allowing any direct traffic inbound from the outside world.

At deployment, a script is executed which includes a configuration file with information such as the external URL of the portal, data directory for scratch and download preparation, as well as the user under which to run the services. The deployment script then pushes this data to the myriad configuration files used by the ODI-PPA stack and places them in the appropriate directory on the host. Individual run scripts automate network creation, and mount the configuration and data directories into each container's local filesystem before initialization.

This architecture allows us to save a great deal of time and energy when performing maintenance or redeploying the ODI-PPA to new hosts, while retaining the flexibility to reconfigure the software stack or move individual components at will. The docker build scripts, and other configuration and run scripts for our ODI DockStream project are available at [6].

2.2.1 Building a Custom Container

Let us look more closely at the ODI-Portal container as a case-in-point. We start by building a base odi-php-zend container which bundles together the base dependencies for this portal architecture:

- The Apache web-server with PHP modules installed.
- The Zend PHP application framework.
- Several PHP libraries.
  - This includes php-pecl-amqp, which itself has ugly dependencies on librabbitmq.so.1 in EL7.
  - Additionally, several more common PHP modules are required for MySQL connectivity, etc.
- Python, and several common modules.

```
$ cat centos-php-zend-odi/dockerfile

# dockerfile to create underlying PHP/Zend on CentOS container
FROM centos/httpd

RUN curl --silent --location https://rpm.nodesource.com/setup_8.x | bash -

RUN yum -y install epel-release
RUN yum -y makecache
RUN yum -y install libssh2 libssh2-devel php-mysql [...] python2-dateutil
COPY librabbitmq-0.5.2-1.el6.x86_64.rpm /root/.
RUN rpm -i /root/librabbitmq-0.5.2-1.el6.x86_64.rpm
RUN yum -y install php-pecl-amqp
CMD ["/run-httpd.sh"]


$ cat centos-php-zend-odi/build.sh

#!/bin/bash
docker build -t centos-php-zend-odi .


$ cat odi-portal/dockerfile

# dockerfile to create ODI portal container
FROM centos-php-zend-odi

ADD create-odiuser /root/create-odiuser
RUN /root/create-odiuser
RUN chown -R foo:foogroup /var/lib/php
RUN /usr/bin/pip install pika pyfits
RUN mkdir /usr/local/portal
COPY odi-portal.tgz /root/odi-portal.tgz
RUN tar xzf /root/odi-portal.tgz -C /usr/local/portal/
RUN ls -lash /usr/local/portal/
ADD functions /root/functions
ADD run-odi.sh /root/run-odi.sh
CMD ["/root/run-odi.sh"]
```

Listing 1. Dockerfiles/scripts to build a custom ODI portal docker container

As this base architecture is used by several other SCA projects, we opted to make this container generic and inheritable by individual portal containers with only minor per-project changes. For ODI-PPA, this amounts to a few minor additions in a separate dockerfile, to create a custom ODI Portal container:

- A script to create Unix users for the project accounts and set up permissions.
- The ODI Portal code itself pulled from github, with additional specific dependencies installed via composer already in place.
- Additional Python modules to deal with FITS images
- A wrapper script to start Apache and the Portal application with it.

We do not package the configuration files for Apache or the Portal itself with the container in order to maintain re-usability. Instead, we have created a run-script for each container which sets some variables at run-time (such as container's IP address on the virtual network) and mounts the needed configuration directories into the new container when it is created. These configurations are accessible from the host and can be adjusted at any time if need be.

With the container up and running, we add a proxy_pass directive to the host's Nginx configuration, allowing web traffic to be passed along to this container. The host's Nginx instance is also responsible for encrypting these connections via SSL. This is described in further detail below.

2.2.2 Deploying Docker Containers

While we could use `docker run` manually to execute our containers when needed, the additional configuration, logging, and data directories needed by the ODI-PPA require a more involved startup mechanism. To this end, we have created a standardized run-script protocol for Docker containers which sets a number of variables pertinent to the individual application, mounts the needed data from the host, and starts the container on the specified virtual network. As a convenience, we have also created a network creation script (see code listing 2) which is sourced by the run-scripts. The script checks for the existence of the virtual network and export some environment variables to make configuration more streamlined.

```
$ cat create_docker_virtual_network.sh

SUBNET=<RFC 1918 subnet>
GATEWAY=<Gateway address within above subnet>
NAME=<Identifier for network>
EXP_SUB=$( echo "${SUBNET}" | cut -d '.' -f1,2,3 )

function create {

docker network rm ${NAME}
docker network create ${NAME} --subnet="${SUBNET}" --gateway="${GATEWAY}" \
        --opt com.docker.network.bridge.enable_icc=true \
        --opt com.docker.network.bridge.ip_forward=true \
        --opt com.docker.network.bridge.enable_ip_masquerade=true \
        --opt com.docker.network.bridge.name=<Virtual NIC name> \
        --opt com.docker.network.driver.mtu=1500
}
if [[ -z `docker network list | grep ${NAME}` ]]; then
    create
else
    echo "Network ${NAME} already exists!"
fi

export <NAME>_SUBNET="${EXP_SUB}"
```

Listing 2. Custom Docker Virtual Network Creation Script

```
#! /bin/bash

SVCNAME=portal
CONTNAME=odi-${SVCNAME}
DOCKERDATA="<Host Storage Path>/${CONTNAME}"
HUBNAME=odi-portal
NET="<Network Identifier>"
CONT_IP="1.2.3.4"

## Create network if necessary
. /create_network.sh

docker rm -f ${CONTNAME}

docker run -t \
    --restart=always \
    --name ${CONTNAME}  \
        --network="${NET}" \
        --ip="${CONT_IP}" \
        -v ${DOCKERDATA}/portal/application/<Config File>:/usr/local/portal/<
   Config File> \
        -v ${DOCKERDATA}/portal_logs:/portal_logs \
        -v ${DOCKERDATA}/etc/httpd/conf:/etc/httpd/conf \
        -v ${DOCKERDATA}/etc/httpd/conf.d:/etc/httpd/conf.d \
        -v ${DOCKERDATA}/var/log/httpd:/var/log/httpd \
        -v ${DOCKERDATA}/usr/local/download:/usr/local/download \
        -v /${DOCKERDATA}/data:/data \
        -d ${HUBNAME}

docker logs -f ${CONTNAME}
```

Listing 3. Example Docker Run-script

2.3 Operational Considerations Including Security

2.3.1 Networking and Firewalls

Any codebase of significant age with a web-facing component will pose an increased security risk as time goes on. This is even more true of complex, distributed software solutions such as the ODI-PPA. While we continue to maintain the code, we did want to limit the attack surface as much as possible. A large portion of this solution hinges on the liberal use of Docker's virtual networks, a proxying web server such as Nginx, and Linux' firewall utilities to isolate the components of this stack as much as possible.

By default, Docker will create a virtual network that is bridged onto the host's primary network interface, and allows ports from containers to be bound directly to the host's external-facing network. While this makes it very easy to stand up a new container running a web-facing service such as Apache, it also means it is very easy to stand up a database server that is open to the world. Thankfully, Docker also allows for the creation of isolated virtual networks using a local-only address space [Figure 1]. This places the host in the role of a gateway router and firewall to direct packets between containers, and restrict access from the outside world.

This allows us to keep non-public portions of the stack (such as the database server) completely unreachable from outside the host, while passing web traffic through an encrypted connection to the appropriate container. While it is possible to accomplish this solely utilizing the NAT functionality within IPTables to mark packets and forward them to the Portal container, this leaves the job of SSL encryption to the webserver running therein and unduly increases the complexity of our deployment. By using Nginx running natively on the host as a proxy server, we are able to cleanly pass encrypted traffic to our Portal with only a single pair of firewall rules.

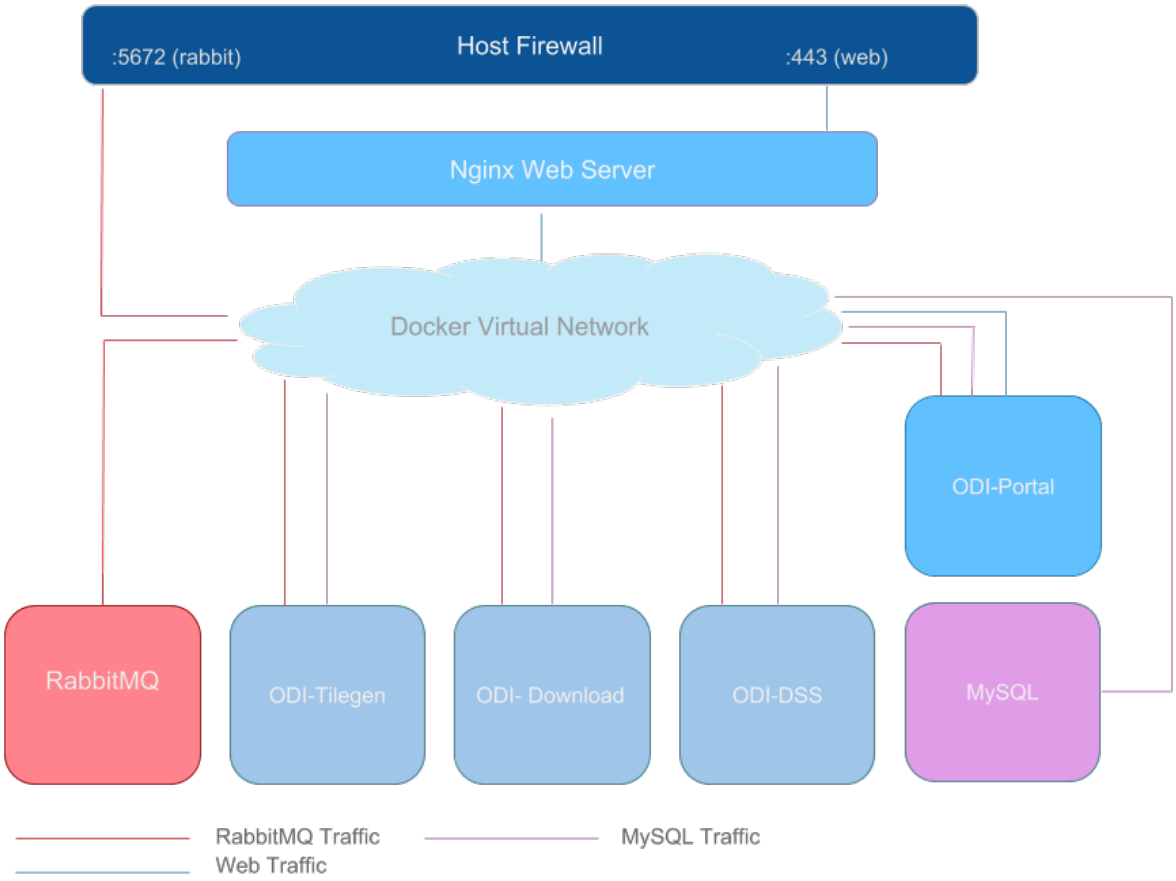

Figure 1. Networking diagram for ODI-PPA deployment in Docker

Such a solution does introduce a small amount of complexity when running containers on disparate hosts: namely, that traffic to the RabbitMQ message broker must be forwarded to the appropriate container by the host's firewall. This is accomplished rather simply by creating a firewall rule which matches packets from the IP addresses used by the other ODI hosts and forwarding port 5672 into the IP address of the RabbitMQ container.

2.3.2 Proxying Web Traffic

As mentioned in the previous section, we rely on a proxying web server to bridge web-facing containers with the outside world. In the particular case of the ODI Portal, the PHP/Zend application requires the Apache web server. As we in the SCA have standardized use of the Nginx webserver elsewhere in our inventory, we are running an instance of Nginx natively on the host to proxy secure traffic from the outside world into the Portal container's Apache instance.

```
[js-156-231] root ~-->grep -B 2 -A 1 "5672" /etc/firewalld/zones/public.xml
  <rule family="ipv4">
    <source ipset="sca_all"/>
    <forward-port to-port="5672" port="5672" protocol="tcp" to-addr="1.2.3.4"
    />
  </rule>
```

Listing 4. Forwarding RabbitMQ traffic from a specific set of IPs (IPSet) into RabbitMQ container

```
server {
    listen 443 ssl http2;
    server_name jsportal.odi.iu.edu;
    access_log  /var/log/nginx/jsportal_ssl_access.log ;
    error_log  /var/log/nginx/jsportal_ssl_error.log;

    ssl     on;
    ssl_certificate /etc/some_dir/cert.pem;
    ssl_certificate_key /etc/some_dir/key.pem;

    # [...] Nominal/current SSL best practice config options
    #        available upon request [...]

    location / {
        proxy_pass http://1.2.3.5:80/;
    }
```

Listing 5. Proxying web traffic into our portal container

The combination of custom Docker containers, operational practices, and networking configurations allows the SCA team to redeploy the ODI-PPA service stack with relative ease, regardless of the underlying physical or virtual hardware, or even on the cloud. We have thus far described the *Dock* part of ODI DockStream, in the next section we describe the *Stream* part of our solution - specifically our use of the Jetstream cloud resource to host the ODI-PPA docker containers for the proof of concept ODI DockStream deployment.

3. TOWARD HARDWARE SUSTAINBILITY

While the SCA team will continue to operate our own hardware solutions to host projects such as the ODI-PPA, such a situation cannot be taken for granted in an arena where strict budgetary guidelines may delay or outright deny the purchase of new hardware. There are of course third-party services such as Amazon AWS or Microsoft Azure, but they incur a cost and are not typically geared directly toward use in a research environment. Moreover, we would face increased difficulty in setting up access to the research file systems and tape archive housed at Indiana University.

The NSF-funded Jetstream [7] is a cloud computing environment designed to provide researchers of the Extreme Science and Engineering Discovery Environment (XSEDE) [8] on-demand access to interactive computing and data analysis resources. Jetstream is supported by Indiana University's Pervasive Technology Institute (PTI), Texas Advanced Computing Center (TACC), and several other partners.

Jetstream [Figure 2] provides a library of pre-tuned virtual machines (VMs) to support scientific analysis and collaboration in a variety disciplines. Generic Linux-based VM configurations are also available, allowing creation of custom VMs to support specific applications or disciplines. For VM creation and management, Jetstream offers the choice of a web-based graphical user interface (called Atmosphere) or direct access to the underlying OpenStack platform via a command line interface. For this paper, we have crafted a lightweight and reusable image with the Nginx webserver and the Docker daemon configured and ready to host the ODI DockStream deployment[†].

3.1 XSEDE Procedures for Requesting Access to Jetstream

Jetstream and XSEDE have a convenient trial allocation option designed to give potential users faster but limited access to its cloud resources, and get a feel for the VM creation and usage process. Without any formal allocation proposals, and within one business day, users are able to evaluate a small 2-core Jetstream VM with limited amount of disk.

[†]By the time our Jetstream allocation runs out, we expect the production deployment of the dockerized ODI-PPA stack to be in place; the ODI DockStream web portal URL will be redirected to the production ODI-PPA portal URL.

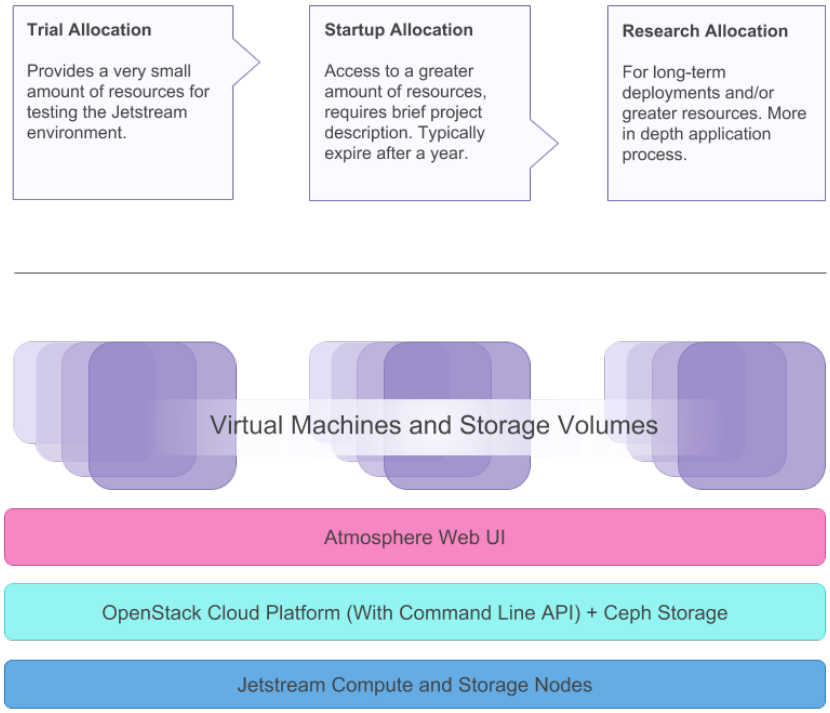

Figure 2. Allocations and Technical Overview of Jetstream

Users needing more resources - also referred to as Service Units or SUs by XSEDE - than what a trial allocation provides can submit a formal startup allocation proposal with a minimal set of required documentation including a project abstract and the Principal Investigator's CV. These are typically allocated for a period of one year that can be extended in certain circumstances. PIs are able to request access for other to their allocation for other users. Startup allocation requests can be submitted throughout the year, and are typically reviewed and awarded within two weeks.

Users considering using Jetstream for long-term operations as well as those needing larger amount of compute and storage resources need to submit a Research Allocation Proposal which entails furnishing a much more detailed project description, benchmarking statistics showing why a certain amount of compute and/or storage is being requested, and a few other supporting documents. These Research Allocation proposals can be submitted 4 times a year and entail a longer review period as well.

Additional technical information about Jetstream is available in [9] while more details about the XSEDE allocation and review process can be found on [10].

4. CONCLUSION

As part of our ongoing strategic planning efforts, we identified risks and potential pitfalls in our One Degree Imager - Portal, Pipeline, and Archive (ODI-PPA) operations. We evaluated possible mitigation strategies, and concluded that leaving the service stack as-is (with ongoing maintenance and bug fixes, and occasional addition of new features), and dockerizing individual services for seamless deployment on one or more cloud-based servers as the optimal path. In this paper, we have described the processes involved into standing up a complete ODI-PPA stack using services running inside docker containers; the creation of such custom containers; and operational considerations that enhance the security and stability of the setup as as a whole. We also described the process of acquiring free-of-cost hardware resources on the Jetstream cloud system.

4.1 Extending Lessons Learned - Related Work

As a software development and operations team at a large and renowned academic institution (Indiana University), the SCA team aspires to apply lessons learned and use best practices derived out of one project in other areas wherever feasible. We recognized that the software sustainability research and development effort for ODI DockStream can and should be applied to other SCA projects, including the concurrently developed ImageX, an imaging archive software stack, described in [11]. While some of that effort is ongoing, we want to highlight and briefly discuss one particular use-case in this paper. IU uses the Moab scheduler to allocate resource time

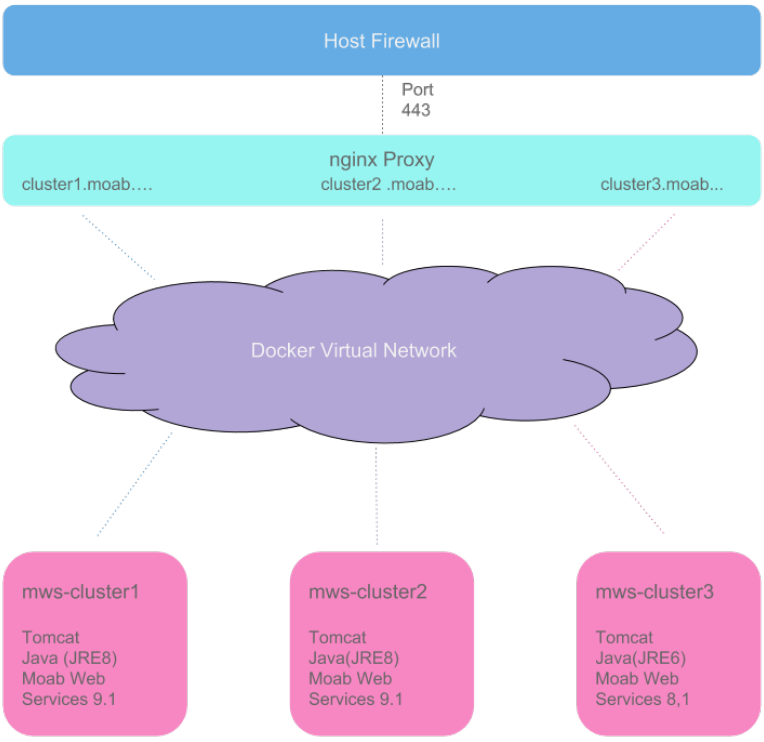

Figure 3. Networking diagram for Moab Web Services (MWS) in Docker

for jobs on its supercomputers and compute clusters. Moab stores a wealth of information which would benefit users of our systems, for e.g., details about all currently queued jobs, status of one's own job submissions, etc. While command-line tools are included with Moab to gather this data, it is not easy for novice users to get the information they need. Moab Web Services (MWS) - an Apache Tomcat based web service - offers a RESTful API to all of the scheduling data.

Deploying this tool posed several challenges: 1) MWS is tightly linked to the version of Moab running on the clusters. As the clusters run different versions of the scheduler, different versions of MWS needed to be run as well. 2) MWS appeared to have certain rigid pre-requisites, for e.g. needing a dated, unmaintained, deprecated and vulnerable version of Java required for the Tomcat container to run successfully. 3) MWS itself has a graphical web user-interface but it is not very intuitive to end-users. Additionally, we were averse to deploying multiple servers to house the multiple MWS instances needed, nor did we want to expose a set of potentially vulnerable services to the external network.

We used the methodology described in this paper to build custom Docker containers for each version of MWS we needed to deploy - including the appropriate version of Java and other dependencies. We then deployed multiple MWS Docker containers with the web service visible only within the custom Docker network. Finally, we proxied the instances of MWS to a public network interface via nginx, and secured it to be only available to a

few servers that make use of the information provided by MWS and then present it in a much more user-friendly web interface within a web application titled HPC Every Where (HEW). Discussing the features of the HEW web interface is beyond the scope of this paper, we expect to publish a separate paper about HEW in the near future.

4.2 Future Work

Our future plans include but are not limited to:

- Deploy a complete production instance of the dockerized ODI-PPA service stack based on our ODI DockStream proof of concept. This is planned for Fall 2018.
- Evaluate migration of other legacy PHP/Zend + Python based SCA systems that are similar to ODI-PPA to a dockerized setup including SpArc [12] and GCS-SCA [13].

ACKNOWLEDGMENTS

We want to express our profound gratitude to all the open source developers who have contributed to Docker, nginx, and other libraries used by ODI-PPA and ODI DockStream.

REFERENCES

[1] Gopu, A., Hayashi, S., Young, M. D., Kotulla, R., Henschel, R., and Harbeck, D. R., "Trident: Scalable compute archive, and visualization/analysis systems," (2016).

[2] Gopu, A., Hayashi, S., Young, M. D., Harbeck, D. R., Boroson, T., Liu, W., Kotulla, R., Shaw, R., Henschel, R., Rajagopal, J., Stobie, E., Knezek, P., Martin, R. P., and Archbold, K., "ODI - Portal, Pipeline, and Archive (ODI-PPA): a web-based astronomical compute archive, visualization, and analysis service," (2014).

[3] "NodeJS JavaScript Runtime." https://nodejs.org.

[4] Merkel, D., "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.* **2014** (Mar. 2014).

[5] "Nginx server." https://nginx.org.

[6] "ODI DockStream Github repository." https://github.com/IUSCA/odi-dockstream.

[7] Stewart, C. A., Cockerill, T. M., Foster, I., Hancock, D., Merchant, N., Skidmore, E., Stanzione, D., Taylor, J., Tuecke, S., Turner, G., Vaughn, M., and Gaffney, N. I., "Jetstream: A self-provisioned, scalable science and engineering cloud environment," in [*Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*], *XSEDE '15*, 29:1–29:8, ACM, New York, NY, USA (2015).

[8] Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G. D., Roskies, R., Scott, J. R., and Wilkins-Diehr, N., "XSEDE: Accelerating Scientific Discovery," *Computing in Science Engineering* **16**, 62–74 (Sept 2014).

[9] "IU Knowledge Base: On XSEDE, what is Jetstream?." https://kb.iu.edu/d/bfde.

[10] "XSEDE User Portal: Research Allocations." https://portal.xsede.org/allocations/research.

[11] Young, M. D., Perigo, R. L., and Gopu, A., "Imagex: a full stack imaging archive solution," (2018).

[12] Pilachowski, C., Hinkle, K., Brokaw, H., Young, M. D., and Gopu, A., "Preservation of 20 years of spectrographic data," (2016).

[13] Young, M. D., Rhode, K., and Gopu, A., "A science portal and archive for extragalactic globular cluster systems data," (2015).