Draft version January 13, 2022
Typeset using LaTeX **twocolumn** style in AASTeX631

The Astronomy Commons Platform: A Deployable Cloud-Based Analysis Platform for Astronomy

Steven Stetzler,[1] Mario Jurić,[1] Kyle Boone,[1] Andrew Connolly,[1] Colin T. Slater,[1] and Petar Zečević[2,3]

[1] DiRAC Institute and the Department of Astronomy, University of Washington, Seattle, USA
[2] Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia
[3] Visiting Fellow, DIRAC Institute, University of Washington, Seattle, USA

ABSTRACT

We present a scalable, cloud-based science platform solution designed to enable next-to-the-data analyses of terabyte-scale astronomical datasets. The presented platform is built on Amazon Web Services (over Kubernetes and S3 abstraction layers), utilizes Apache Spark and the Astronomy eXtensions for Spark for parallel data analysis and manipulation, and provides the familiar JupyterHub web-accessible front-end for user access. We outline the architecture of the analysis platform, provide implementation details, rationale for (and against) technology choices, verify scalability through strong and weak scaling tests, and demonstrate usability through an example science analysis of data from the Zwicky Transient Facility's 1Bn+ light-curve dataset. Furthermore, we show how this system enables an end-user to iteratively build analyses (in Python) that transparently scale processing with no need for end-user interaction.

The system is designed to be deployable by astronomers with moderate cloud engineering knowledge, or (ideally) IT groups. Over the past three years, it has been utilized to build science platforms for the DiRAC Institute, the ZTF partnership, the LSST Solar System Science Collaboration, the LSST Interdisciplinary Network for Collaboration and Computing, as well as for numerous short-term events (with over 100 simultaneous users). A live demo instance, the deployment scripts, source code, and cost calculators are accessible at http://hub.astronomycommons.org/.

Keywords: Cloud computing (1970) — Astronomy data analysis (1858) — Astronomy databases (83) — Light curves (918)

1. INTRODUCTION

Today's astronomy is undergoing a major change. Historically a data-starved science, it is being rapidly transformed by the advent of large, automated, digital sky surveys into a field where terabyte and petabyte data sets are routinely collected and made available to researchers across the globe.

The Zwicky Transient Facility (ZTF; Bellm et al. 2019; Graham et al. 2019; Dekany et al. 2020; Masci et al. 2019) has engaged in a three-year mission to monitor the Northern sky. With a large camera mounted on the Samuel Oschin 48-inch Schmidt telescope at Palomar Observatory, the ZTF is able to monitor the entire visible sky almost twice a night. Generating about 30 GB of nightly imaging, ZTF detects up to 1,000,000 variable, transient, or moving sources (or alerts) every night, and makes them available to the astronomical community (Patterson et al. 2018). Towards the middle of 2024, a new survey, the Legacy Survey of Space and Time (LSST; Ivezić et al. 2019), will start operations on the NSF Vera C. Rubin Observatory. Rubin Observatory's telescope has a mirror almost seven times larger than that of the ZTF, which will enable it to search for fainter and more distant sources. Situated in northern Chile, the LSST will survey the southern sky taking $\sim$1,000 images per night with a 3.2 billion-pixel camera with a $\sim$10 deg^2 field of view. The stream of imaging data ($\sim$6PB/yr) collected by the LSST will yield repeated measurements ($\sim$100/yr) of over 37 billion objects, for a total of over 30 trillion measurements by the end of the next decade. These are just two examples,

Corresponding author: Steven Stetzler
stevengs@uw.edu

with many others at similar scale either in progress (Kepler, Pan-STARRS, DES, GAIA, ATLAS, ASAS-SN; Kaiser et al. 2010; Dark Energy Survey Collaboration et al. 2016; Gaia Collaboration et al. 2016; Tonry et al. 2018; Shappee et al. 2014) or planned (WFIRST, Euclid; Spergel et al. 2015; Scaramella et al. 2014). They are being complemented by numerous smaller projects ($\lesssim$\$1M scale), contributing billions of more specialized measurements.

This 10-100x increase in survey data output has not been followed by commensurate improvements in tools and platforms available to astronomers to manage and analyze those datasets. Most survey-based studies today are performed by navigating to archive websites, entering (very selective) filtering criteria to download "small" ($\sim$10s of millions of rows; $\sim$10GB) subsets of catalog products. Those subsets are then stored locally and analyzed using custom routines written in high-level languages (e.g., Python or IDL), with the algorithms generally assuming in-memory operation. With the increase in data volumes and subsets of interest growing towards the $\sim$100GB-1TB range, this mode of analysis is becoming infeasible.

One solution is to provide astronomers with access to the data through web portals and *science platforms* – rich gateways exposing server-side code editing, management, execution and result visualization capabilities – usually implemented as notebooks such as Jupyter (Kluyver et al. 2016) or Zeppelin (Cheng et al. 2018). These systems are said to *bring the code to the data*, by enabling computation on computational resources co-located with the datasets and providing built-in tools to ease the process of analysis. For example, the LSST has designed (Jurić et al. 2017; Dubois-Felsmann et al. 2017) and implemented a science platform suitable for their use cases based on the ability to do all work remotely through a web-browser.[1] While such science platforms are a major step forward in working with large datasets, they still have some limitations. For example, platforms that are deployed on traditional HPC systems or on on-premises hardware can suffer from having insufficient computing next to the data: all users of shared HPC resources are familiar with "waiting in the queue" due to over subscription. Science platforms built on cloud computing resources will find it much easier to provide computing resources according to user demand: this is the promise of "elastic" computing in the cloud.

Secondly, even when surveys deploy distributed SQL databases for serving user queries (e.g. Qserv in the case of LSST; Wang et al. 2011), user analysis is still not easily parallelized – query requests and results are bottlenecked at one access point which severely limits scalability. In contrast, the system we describe and implement provides direct, distributed access to data for a user's analysis code. Finally, current science platforms do not tackle the issue of working on multiple large datasets at the same time – if they're in different archives, they still have to be staged to the same place before work can be done. In other words, they continue to suffer from availability of computing, being I/O-bound, and geographic dislocation.

We therefore need to not only bring the code to the data, but also *bring the data together*, co-locate it next to an (ideally limitless) reservoir of computing capacity, with I/O capabilities that can scale accordingly. Furthermore, we need to make this system *usable*, by providing astronomer-friendly frameworks for working with extremely large datasets in a scalable fashion. Finally, we need to provide a user-interface which is accessible and familiar, with a shallow learning curve.

We address the first of these challenges by utilizing the Cloud (in our case, Amazon Web Services) to supply data storage capacity and effective dataset co-location, I/O bandwidth, and (elastic) compute capability. We address the second challenge by extending the Astronomy eXtensions for Spark (AXS; Zečević et al. 2019), a distributed database and map-reduce like workflow system built on the industry-standard Apache Spark (Zaharia et al. 2010) engine, to work in this cloud environment.[2] Spark allows the execution of everything from simple ANSI SQL-2011 compliant queries, to complex distributed workflows, all driven from Python. Next, we build a JupyterHub facade as the entry-point to the system. Finally, we make it possible for IT groups (or advanced users) to easily deploy this entire system for use within their departments, as an out-of-the-box solution for cloud-based astronomical data analysis.

The combination of these technologies allows the researcher to migrate "classic" subset-download-analyze workflows with little to no learning curve, while providing an upgrade path towards large-scale analysis. We validate the approach by deploying the ZTF dataset (a precursor to LSST) on this system, and demonstrate it can be successfully used for exploratory science.

2. A PLATFORM FOR USER-FRIENDLY SCALABLE ANALYSIS OF LARGE ASTRONOMICAL DATASETS

[1] See https://data.lsst.cloud/

[2] See https://spark.apache.org/

We begin by introducing the properties of cloud systems that make them especially suitable for scalable astronomical analysis platforms, discuss the overall architecture of our platform, its individual components, and performance.

2.1. *The Cloud*

Traditionally, computing infrastructure was acquired and maintained close to the group utilizing the resource. For example, a group led by a faculty member would purchase and set up one or more machines for a particular problem, or (on a larger scale) a university may centralize computing resources into a common cluster, shared with the larger campus community. These acquisitions – so-called "on-premise" computing – are capital heavy (require a large initial investment), require local IT knowledge, and allow for a limited variety of the systems being purchased (e.g., a generic Linux machine for a small group, or standardized types of nodes for an HPC cluster).

Cloud services move this infrastructure (and the work to maintain it) away from the user, and centralize it with the cloud provider. The infrastructure is provided as a service: individual machines, entire HPC clusters, as well as higher-order services (databases, filesystes, etc.) are *rented* for the time the resource is needed, rather than purchased.

They are billed proportional to usage; virtual machines are typically rented by the second, virtual networks priced by bandwidth usage, and virtual storage priced by storage size per unit time. These components are provisioned by the user on-demand, and are built to be "elastic." One can typically rent several hundred virtual machines and provision terabytes of storage space with an expectation that it will be delivered within minutes and then release these resource back to the cloud provider at will. This usage and pricing model offers the unique benefit of providing access to affordable computing at scale. One can rent hundreds of virtual machines for a short period of time (just the execution time of a science workflow) without investing in the long-term support of the underlying infrastructure. In addition, cloud providers typically offer managed storage solutions to support reading/writing data to/from all of these machines. These so-called "object stores" are highly available, highly durable, and highly scalable stores of arbitrarily large data volumes. For example, Amazon Simple Storage Solution (Amazon S3) provides scalable, simultaneous access to data through a simple API over a network.[3] S3 supports very high throughput at the terabit-per-second assuming storage access patterns are optimized.[4] Once a solution for scalable storage is added to the mix, cloud computing systems start to resemble the traditional supercomputers many scientists are already familiar with for running simulations and performing large-scale data analysis.

2.2. *Orchestrating cloud applications: Kubernetes*

The pain point that remains in managing and developing applications for the cloud is the problem of orchestration: it can become burdensome to write custom software for provisioning and managing cloud resources, and there is a danger of cloud "lock-in" occurring when software applications become too strongly coupled with the cloud provider's API. The open source community has developed orchestration tools, like Kubernetes, to address this issue.[5]

Kubernetes is used to schedule software applications packaged in Docker images and run as Docker containers on a cluster of computers while handling requests for and the provisioning of cloud resources to support running those containers.[6] Kubernetes provides a cloud-agnostic API to describe cloud resources as REST objects.[7] Storage is described using "Persistent Volume" objects, requests for that storage using "Persistent Volume Claim" objects, and networking utilities like routing, port-forwarding, and load balancing using "Service" objects. A single application is specified using a "Pod" object that references storage objects and service objects by name to link an application to these resources. In addition, the Pod object allows one to impose CPU and memory limits on an application or assign the application to a certain node, among other features.

The Kubernetes control plane handles provisioning of hardware from the cloud provider to satisfy the requirements of its objects. For example on AWS, an outstanding request for a Service requiring a load balancer will be fulfilled by creating an AWS Elastic Load Balancer (ELB) or Application Load Balancer (ALB). Simi-

[3] Amazon S3 uses a REST API with HTTP.

[4] This is detailed in the S3 documentation: https://docs.aws.amazon.com/AmazonS3/latest/dev/optimizing-performance.html

[5] The Kubernetes documentation provides a thorough and beginner-friendly introduction to the software: https://kubernetes.io/docs/

[6] Docker isolates software programs at the level of the operating system, in contrast to virtual machines which isolate operating systems from one another at the hardware level. See https://www.docker.com/ and https://docs.docker.com/ for more information.

[7] REST refers to "representational state transfer," a style of software architecture that is ubiquitous in modern software, especially on the web.

Network

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-load-balancer
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 8888
    protocol: TCP
    name: http
  selector:
    app: notebook-pod
```

Storage

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-volume-claim
spec:
  resources:
    requests:
      storage: 10Gi
      storageClassName: ebs
```

Application

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jupyter-notebook
spec:
  replicas: 1
  selector:
    matchLabels:
      app: notebook-pod
  template:
    metadata:
      labels:
        app: notebook-pod
    spec:
      containers:
      - name: notebook
        image: jupyter/scipy-notebook
        ports:
        - containerPort: 8888
        volumeMounts:
        - mountPath: "/home/jovyan"
          name: my-volume
      volumes:
      - name: my-volume
        persistentVolumeClaim:
          claimName: my-volume-claim
```

Figure 1. An illustration of the structure and composition of YAML-formatted text specifying Kubernetes objects that together create a functional and internet-accessible Jupyter notebook server. The Jupyter notebook application is created as a Pod on the cluster (right). Networking objects (top left) specify how a public-facing load balancer can be connected to the Jupyter notebook Pod (`notebook-pod`) on a certain port (`8888`). Storage objects trigger the creation of, for example, hard drive disk space from the cloud provider (bottom left). Colored text indicate how the files are linked to support one another: blue indicates how network and application are linked, orange how application and storage are linked, and green how storage volumes are mounted into the filesystem of the application.

larly, an outstanding request for a Persistent Volume will be fulfilled by creating an Amazon Elastic Block Store (EBS) volume. The handling of hardware provisioning in the control plane decouples software applications from the cloud whose hardware they run on.

Each Kubernetes object is described using YAML, a human-readable format for storing configuration information (lists and dictionaries of strings and numbers).[8] Figure 1 shows an example set of YAML-formatted text describing Kubernetes objects that together would link a Jupyter notebook server backed by a 10 GiB storage device to an internet-accessible URL.[9]

Cloud systems offer unique infrastructure elements that help support a system for scalable science analysis. Virtual machines can be rented in the hundreds or thousands to support large computations, each accessing large datasets in a scalable manner from a managed service. Orchestration layers, like Kubernetes, ease the process of running science software on cloud resources. In section 2.3, we discuss how we leverage cloud infrastructure to build such a platform.

2.3. *System Architecture*

Underlying this platform are four key components:

1. An interface for computing. We use the Jupyter ecosystem, a JupyterHub deployment based on the `zero-to-jupyterhub` project that creates Jupyter notebook servers on our computing infrastructure for authenticated users. A Jupyter notebook server provides a web-interface to interactively run code on a remote machine alongside a set of pre-installed software libraries.[10]

2. A scalable analytics engine. We use Apache Spark, an industry standard tool for distributed data querying and analysis, and the Astronomy eXtensions to Spark (AXS).

3. A scalable storage solution. We use Amazon Simple Storage Solution (S3). Amazon S3 is a managed object store that can store arbitrarily large data volumes and scale to an arbitrarily large number of requests for this data.

4. A deployment solution. We've developed a set of Helm charts and bash scripts automating the deployment of this system onto the AWS cloud. We plan to generalize these to other cloud providers in the future.[11]

Each of these components are largely disconnected from one another and can be mixed and matched with other drop-in solutions.[12] Aside from the deployment solution, each of these components are comprised of simple processes communicating with each other through

[8] See https://yaml.org/ for specification and implementations.

[9] Please see the Kubernetes documentation for further explanation of Kubernetes objects: https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

[10] See https://zero-to-jupyterhub.readthedocs.io/ and https://github.com/jupyterhub/zero-to-jupyterhub-k8s.

[11] See https://helm.sh/

[12] Zepplin notebooks, among other tools, compete with Jupyter notebooks for accessing remote computers for analysis and data visualization. Dask is a competing drop-in for Apache Spark that scales Python code natively. A Lustre file system could be a drop-in for Amazon S3. Amazon EFS, a managed and scalable network filesystem, is also an option. Kustomize is an alternative to Helm.

an API over a network. This means that each solution for (1), (2), and (3) is largely agnostic to the choice of running on a bare-metal machine, inside a virtual machine (VM), inside a Linux container, or using a managed cloud service as long as each component is properly networked.

2.3.1. *An Interface to Computing*

The Jupyter notebook application, and its extension Jupyter lab, provide an ideal environment for astronomers to access, manipulate, and visualize data sets. The Jupyter notebook/lab applications, although usually run locally on a user's machine, can run on a remote machine and be accessed through a JupyterHub, a web application that securely forwards authenticated requests directed at a central URL to a running notebook server.[13] The authentication layer of JupyterHub allows us to block non-authenticated users from the platform. Our science platform integrates authentication through GitHub, allowing us to authenticate both individual users by their GitHub usernames and groups of users through GitHub Organization membership. For example, the implementation of this science platform described in Section 3 restricts access to the platform and its private data to members of the `dirac-institute`[14] and `ZwickyTransientFacility`[15] GitHub organizations.

2.3.2. *A Scalable Analytics Engine*

Apache Spark (Spark) is a tool for general distributed computing, with a focus on querying and transforming large amounts of data, that works well in a shared-nothing, distributed computing environment. Spark uses a driver/executor model for executing queries. The driver process splits a given query into several (1 to thousands) independent tasks which are distributed to independent executor processes. The driver process keeps track of the state of the query, maintains communication with its executors, and coalesces the results of finished tasks. Since the driver and executor(s) only need to communicate with each other over the network, executor processes can remain on the same machine as a driver, to take advantage of parallelism on a single machine, or be distributed across several other machine in a distributed computing context.[16] The API for data transformation, queries, and analysis remains the same whether or not the Spark engine executes the code sequentially on a local machine or in parallel on distributed machines, allowing code that works on a laptop to naturally scale to a cluster of computers.

To support astronomy-specific operations, Zečević et al. (2019) have developed the Astronomy eXtensions to Spark (AXS), a set of additional Python bindings to the Spark API to ease astronomy-specific data queries such as cross matches and sky maps in addition to an internal optimization for speeding up catalog cross matches using the ZONES algorithm, described in Zečević et al. (2019). We include AXS in our science platform to ease the use of Spark for astronomers.

2.3.3. *A Scalable Storage Solution*

Amazon S3 is a scalable object store with built-in backups and optional replication across geographically distinct AWS regions. Files are placed into a S3 bucket, a flat file system that scales well to simultaneous access from thousands of individual clients. The semantics of the S3 API are not compliant with the POSIX specification, a requirement for some use-cases.[17] Additionally, there is no limit to the amount of data that can be stored.[18] We use S3 to store data in Apache Parquet format,[19] a compressed column-oriented data storage format. The columnar nature and partitioning of the files in Parquet format allows for very fast reads of large tables. For example, one can obtain a subset of just the "RA" column of a catalog without scanning through all parts of all of the files.

2.3.4. *A deployment solution*

We have created a deployment solution for organized creation and management of each of these three components. The code for this is stored at a GitHub repository accessible at https://github.com/astronomy-commons/science-platform. Files referenced in the following code snippets assume access at the root level of this repository.

[13] As an example, one may access a JupyterHub at the URL https://⟨hub_url⟩.com which, if you are an authenticated user, will forward through a proxy to https://⟨hub_url⟩.com/user/⟨username⟩. When running a notebook on a local machine, there is no access to a JupyterHub and the single user server is served at (typically) http://localhost:8888.

[14] http://github.com/dirac-institute/

[15] http://github.com/ZwickyTransientFacility

[16] Creating executor processes on a single machine isn't done in practice; instead, Spark supports multithreading in the driver process that replace the external executor process(es) when using local resources.

[17] Projects such as `s3fs` (https://github.com/s3fs-fuse/s3fs-fuse) provide an interface layer between a client and S3 to make the filesystem largely POSIX compliant.

[18] Although individual files must be no larger than 5 TB, and individual PUT requests (upload actions) cannot exceed 5 GB

[19] See https://parquet.apache.org/

To create and manage our Kubernetes cluster, we use the `eksctl` software.[20] This software defines configuration of the Amazon Elastic Kubernetes Service (EKS) from YAML-formatted files. An EKS cluster consists of a managed Kubernetes master node along with a set of either managed or unmanaged nodegroups backed by Amazon Elastic Compute Cloud (EC2) virtual machines which run scheduled containers.[21] The configuration files bundled with our source code generate an EKS cluster along with a set of two managed nodegroups. With the version of the code released with this manuscript, one can create a cluster as follows, running in a Bash shell:

```
$ eksctl create cluster -f ./cluster/eksctl_config.yaml
```

To help us manage large numbers of Kubernetes objects, we use Helm, the "package manager for Kubernetes." Helm allows Kubernetes objects described as YAML files to be templated using a small number of parameters or "values," also stored in YAML. Helm packages together YAML template files and their default template values in Helm "charts." Helm charts can have versioned dependencies on other Helm charts to compose larger charts from smaller ones.

We have created a Helm chart to manage and distribute versioned deployments of our platform. This chart depends on four sub-charts:

1. The `zero-to-jupyterhub` chart, a standard and customizable installation of JupyterHub on Kubernetes. The `zero-to-jupyterhub` chart uses Docker images from the Jupyter Docker Stacks[22] by default and uses the `KubeSpawner`[23] for creating Jupyter notebook servers using the Kubernetes API directly instead of using Helm.

2. The `nfs-server-provisioner` chart, which provides a network filesystem server and Kubernetes-compliant storage provisioner.[24]

3. A `mariadb` chart, which provides a MariaDB server[25] and is used as an Apache Hive metadata store for AXS.[26]

4. The `cluster-autoscaler-chart`, which deploys the Kubernetes Cluster Autoscaler, an application that scales the number of nodes in the Kubernetes cluster up or down when resources are too constrained or underutilized.[27]

In the version of the code released with this manuscript, our published Helm chart can be deployed on a Kubernetes cluster using a single Bash script:

```
$ export NAMESPACE=hub
$ export RELEASE=hub
$ ./scripts/deploy.sh
```

Figure 2 shows the state of the Kubernetes cluster during normal usage of a platform created with our Helm chart as well as the pathway of API interactions that occur as a user interacts with the system. A user gains access to the system through a JupyterHub, which is a log-in portal and proxy to one or more managed Jupyter notebook servers spawned by the JupyterHub. This notebook server is run on a node of the Kubernetes cluster, which can be constrained by hardware requirements and/or administrator provided node labels. A proxy forwards external authenticated requests from the internet to a user's notebook server. Users can use the Apache Spark software, which is pre-installed on their server, to create a Spark cluster using the Spark on Kubernetes API.

2.4. *Providing a shared filesystem with granular access control*

We found it to be critically important to provide a way for users to easily share files with one another. The default Helm chart and `KubeSpawner` configuration creates a Persistent Volume Claim backed by the default storage device configured for the Kubernetes cluster for each single user server, allowing a user's files to persist beyond the lifetime of their server. For AWS, the default storage device is an EBS volume, roughly equivalent to a network-connected SSD with guaranteed input/output capabilities. By default, this volume is mounted at the file system location `/home/jovyan` in the single user container. This setup makes it difficult for the users' results to be shared with others: a) they are isolated

[20] See https://eksctl.io/

[21] Managed nodes are EC2 virtual machines with a tighter coupling to an EKS cluster. Unmanaged nodes allow for more configuration by an administrator.

[22] https://jupyter-docker-stacks.readthedocs.io/

[23] https://jupyterhub-kubespawner.readthedocs.io/

[24] See https://github.com/helm/charts/tree/master/stable/nfs-server-provisioner

[25] See https://mariadb.org/

[26] See https://hive.apache.org/

[27] https://github.com/kubernetes/autoscaler

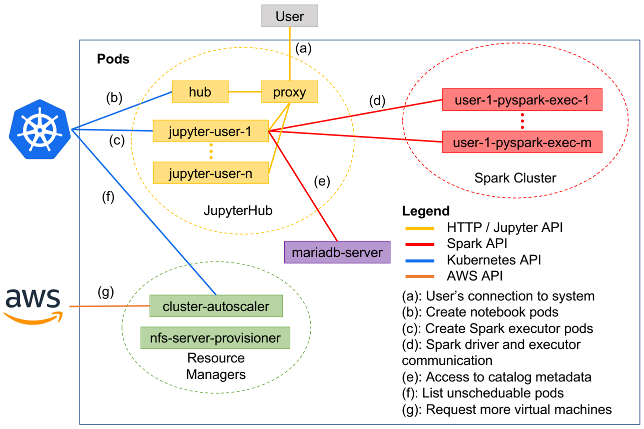

Figure 2. A diagram of the essential components of the Kubernetes cluster when the science platform is in use. Each box represents a single Kubernetes Pod scheduled on the cluster. The colors of the boxes and the dashed ovals surrounding the three groups are for visualization purposes only; each Pod exists as an independent entity to be scheduled on any available machines. The colored paths and letter markers indicate the pattern of API interactions that occur when users interact with the system. (a) shows a user connecting to the JupyterHub from the internet. The JupyterHub creates a notebook server (`jupyter-user-1`) for the user (b). The user creates a Spark cluster using their notebook server as the location for the Spark driver process (c). Scheduled Spark executor Pods connect back to the Spark driver process running in the notebook server (d). The Spark driver process accesses a MariaDB server for catalog metadata (e). In the background, the Kubernetes cluster autoscaler keeps track of the scheduling status of all Pods (f). At any point in (a)-(d), if a Pod cannot be scheduled due to a lack of cluster resources, the cluster autoscaler will request more machines from AWS to meet that need (g).

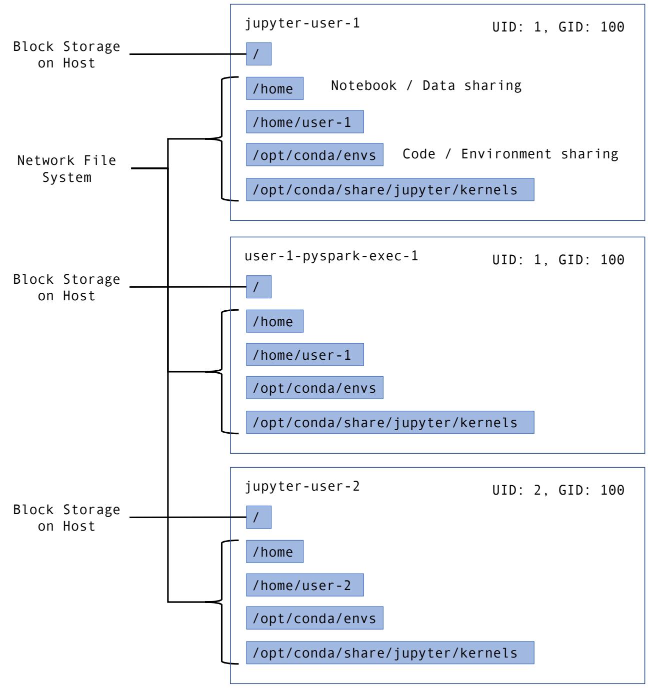

Figure 3. An illustration of the filesystem within each container spawned by the JupyterHub (`jupyter-user-1` and `jupyter-user-2`) and by the user in the creation of a distributed Spark cluster. Most of the filesystem (the root directory `/`) exists on an ephemeral storage device tied to the host machine. The home directories, `conda` environment directories, and Jupyter kernel directories within each container are mounted from an external NFS server. This file structure allows for sharing of Jupyter Notebook files and code environments with other users and with a user's individual Spark Cluster. UNIX user ids (`UID`) and group ids (`GID`) are set to prevent unauthorized data access and edits.

to their own disk, and b) by default all users share the same username and IDs, making granular access control extremely difficult.

To resolve these issues, we provisioned a network file system (NFSv4) server using the `nfs-server-provisioner` Helm chart, creating a centralized location for user files and enabling file sharing between users. To solve the problem of access control, each notebook container is started with two environment variables: `NB_USER` set equal to the user's GitHub username, and `NB_UID` set equal to the user's GitHub user id. The start-up scripts included in the default Jupyter notebook Docker image use the values of these environment variables to create a new Linux user, move the home directory location, update home directory ownership, and update home directory permissions from their default values. Figure 3 shows how the NFS server is mounted into single user pods to enable file sharing. The NFS server is mounted at the `/home` directory on the single user server, and a directory is created for the user at the location `/home/<username>`. Each user's directory is protected using UNIX-level file permissions that prevent other users from making unauthorized edits to their files. System administrators can elevate their own permissions (and access the back-end infrastructure arbitrarily) to edit user files at will. The UNIX user ids (UIDs) are globally unique, since they are equal to a unique GitHub ID.

In initial experiments, we used the managed AWS Elastic File System (EFS) service to enable file sharing. Using the managed service provides significant benefits, including unlimited storage, scalable access, and automatic back-ups. However, EFS had a noticeable latency increase per Input/Ouput operation compared to the EBS-backed storage of the Kubernetes-managed NFS server. In addition, EFS storage is 3× more expensive than EBS storage.[28]

In addition to storing home directories on the NFS server, we have an option to store all of the science analysis code (typically managed as `conda` environments) on the NFS server. This has several advantages relative to the common practice of storing the code into Jupyter notebook Docker images. The primary advantage is that this allows for updating of installed software in real-time, and without the need to re-start user servers. A secondary advantage is that the Docker images become smaller and faster to download and start up (thus improving the user experience). The downside is in decreased scalability: the NFS server includes a central point, shared by all users of the system. Analysis codes are often made up of thousands of small files, and a request for each file when starting a notebook can lead to large loads on the NFS server. This load increases when serving more than one client, and may not be a scalable beyond serving a few hundred users.

For systems requiring significant scalability, a hybrid approach of providing a base `conda` environment in the Docker image itself in addition to mounting user-created and user-managed `conda` environments and Jupyter kernels from the NFS server is warranted. This allows for fast and scalable access to the base environment while also providing the benefit of shared code bases that can be updated in-place by individual users.

2.5. *Providing Optimal and Specialized Resources*

Some users require additional flexibility in the hardware available to match their computing needs. To accommodate this, we have made deployments of this system that allow users to run their notebooks on machines with more CPU or RAM or with specialty hardware like Graphics Processing Units (GPUs) as they require. This functionality is restricted to deployments where we trust the discretion of the users and is not included in the demonstration deployment accompanying this manuscript.

Flexibility in hardware is provided through a custom JupyterHub options form that is shown to the user when they try to start their server. An example form is shown in Fig. 4. Several categories of AWS EC2 instances are enumerated with their hardware and costs listed. Hardware is provisioned in terms of vCPU, or "virtual CPU," roughly equivalent to one thread on a hyperthreaded

[28] The cost of EFS is \$0.30/GB-Month vs \$0.10/GB-Month for EBS. Lifecycle management policies for EFS that move infrequently used data to a higher-latency access tier can reduce costs to approximately the EBS level.

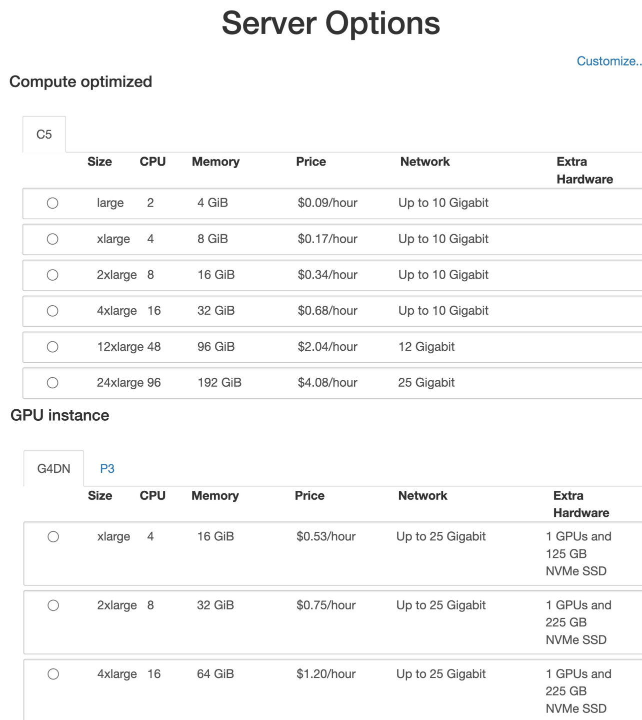

Figure 4. A screenshot of the JupyterHub server spawn page. Several options for computing hardware are presented to the user with their hardware and costs enumerated. Of note is the ability to spawn GPU instances on demand. When a user selects one of these options, their spawned Kubernetes Pod is tagged so that it can only be scheduled on a node with the desired hardware. If a node with the required hardware does not exist in the Kubernetes cluster, the cluster autoscaler will provision it from the cloud provider (introducing a ∼5 minute spawn time).

CPU. In this example, users can pick an instance that has as few resources as 2 vCPU and 1 GiB of memory at the lowest cost of \$0.01/hour (the `t3.micro` EC2 instance), to a large-memory machine with 96 vCPU and 768 GiB of memory at a much larger cost of \$6.05/hour (the `r5.24xlarge` EC2 instance). In addition, nodes with GPU hardware are provided as an option at moderate cost (4 vCPU, 16 GiB memory, 1 NVIDIA Tesla P4 GPU at \$0.53/hour; the `g4dn.xlarge` EC2 instance). These GPU nodes can be used to accelerate code in certain applications such as image processing and machine learning. For this deployment, the form is configured to default to a modest choice with 4 vCPU and 16 GiB of memory at a cost of \$0.17/hour (the `t3.xlarge` EC2 instance). This range of hardware options and prices will change over time; the list provided is simply an example of the on-demand heterogeneity provided via AWS.

3. A DEPLOYMENT FOR ZTF ANALYSES

Name	Data Size (GB)	# Objects (10^9)
SDSS	65	0.77
AllWISE	349	0.81
Pan-STARRS 1	402	2.2
Gaia DR2	421	1.8
ZTF	4100	1.2
Total	5337	8.9

Table 1. The sizes of each of the datasets available on the ZTF science platform along with the total data volume.

To demonstrate the capabilities of our system and verify its utility to a science user, we deployed it to enable the analysis of data from the Zwicky Transient Facility (ZTF). Section 3.1 describes the datasets available through this deployment, Section 3.2 demonstrates the typical access pattern to the data using the AXS API, and Section 3.3 showcases a science project executed on this platform.

3.1. *Datasets available*

Table 1 enumerates the datasets available to the user in this example deployment. We provide de-duplicated ZTF match files for analysis of light curves of objects detected by ZTF. The most recent version of these match files have a data volume of $\sim$ 4 TB describing light curves of $\sim$ 1 billion+ objects in the "g", "r", and "i" bands. In addition, we provide access to the data releases from the SDSS, Gaia, AllWISE, and Pan-STARRS surveys for convenient cross matching of ZTF to other datasets.[29]

3.2. *Typical workflow*

Data querying is available to the user through the AXS/Spark Python API. These data are accessed through the AXS/Spark Python API in a simple manner. Data loading follows a pattern like:

```
import axs
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
catalog = axs.AxsCatalog(spark)
ztf = catalog.load('ztf')
```

The `spark` object represents a Spark SQL Session connected to a Hive metastore database where the data have already been ingested. This is passed to the `AxsCatalog` object to use as a SQL backend. Catalogs from the metastore database are loaded by name using the AXS API. Data subsets can be created by selecting one or more columns:

```
ztf_subset = ztf.select('ra', 'dec', 'mag_r')
```

`AxsCatalog` Python objects can be crossmatched with one another to produce a new catalog with the crossmatch result:

```
gaia = catalog.load('gaia')
xmatch = ztf.crossmatch(gaia)
```

The `xmatch` object can be queried like any other `AxsCatalog` object. Spark allows for the creation of User-Defined Functions (UDFs) that can be mapped onto rows of a Spark DataFrame. The following example shows how a Python function that converts an AB magnitude to its corresponding flux in janskys can be mapped onto all $\sim$63 billion r-band magnitude measurements from $\sim$1 billion light curves in the ZTF dataset (in parallel):

```
from pyspark.sql.functions import udf
from pyspark.sql.types import ArrayType
from pyspark.sql.types import FloatType
import numpy as np

@udf(returnType=ArrayType(FloatType()))
def abMagToFlux(m):
    flux = ((8.90 - np.array(m))/2.5)**10
    return flux.tolist()
ztf_flux_r = ztf.select(
    abMagToFlux(ztf['mag_r']).alias("flux_r")
)
```

3.3. *Science case: Searching for Boyajian star Analogues*

We test the ability of this platform to enable large-scale analysis by using it to search for Boyajian star (Boyajian et al. 2016) analogs in the ZTF dataset. The Boyajian star, discovered with the Kepler telescope, dips in its brightness in an unusual way. We intend to search the ZTF dataset for Boyajian-analogs, other stars that have anomalous dimming events, which will be fully described in Boone et al. (in prep.); here we limit ourselves to aspects necessary for the validation of the analysis system. The main method for our Boyajian-analog searches relies on querying and filtering large volumes of ZTF light curves using AXS and Apache Spark in search of the dimming events. This presents an ideal science-case for our platform: the *entire ZTF dataset* must be queried, filtered, and analyzed repeatedly in order to complete the science goals.

We wrote custom Spark queries that search the ZTF dataset for dimming events. After filtering of the data,

[29] Other tabular data can be added to the system by the user. Additional data products from these surveys, such as images, can be stored and accessed with AXS.

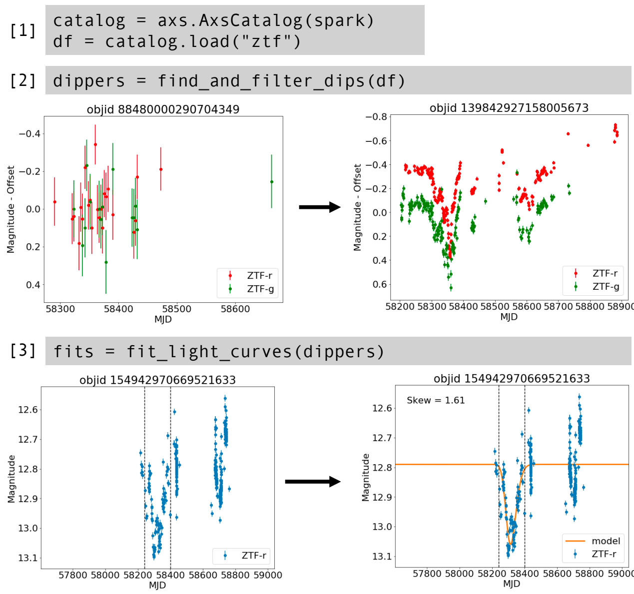

Figure 5. An example analysis (boiled down to two lines) that finds light curves in the ZTF dataset with a dimming event. (1) shows how the ZTF dataset is loaded as a Spark DataFrame (`df`), (2) shows the product of filtering light curves for dimming events, and (3) shows the result of fitting a model to the remaining light curves. This process exemplifies that analyses can often be represented as a filtering and transformation of a larger dataset, a process that Spark can easily execute in parallel.

we created a set of UDFs for model fitting that wrap the optimization library from the `scipy` package. These UDFs are applied to the filtered lightcurves to parallelize least-squared fitting routines of various models to the dipping events. Figure 5 shows an outline of this science process using AXS.

The use of Apache Spark speeds up queries, filtering, and fitting of the data tremendously when deployed in a distributed environment. We used a Jupyter notebook on our platform to allocate a Spark cluster of consisting of 96 `t3.2xlarge` EC2 instances. Each instance had access to 8 threads running on an Intel Xeon Platinum 8000 series processor with 32 GiB of RAM, creating a cluster with 768 threads and 3,072 GiB of RAM. We used the Spark cluster to complete a complex filtering task on the full 4 TB ZTF data volume in ∼three hours. The underlying system was able to scale to full capacity within minutes, and scale down once the demanding query was completed just as fast, providing extreme levels of parallelism at minimal cost. The total cost over the time of the query was ∼\$100.

This same complex query was previously performed on a large shared-memory machine at the University of Washington with two AMD EPYC 7401 processors and 1,024 GiB of RAM. The query utilized 40 threads and accessed the dataset from directly connected SSDs. This query previously took a full two days to execute on this hardware in comparison to the ∼three hours on the cloud based science platform. Performing an analysis of this scale would not be feasible if performed on a user's laptop using data queried over the internet from the ZTF archive.

In addition, the group was able to gain the extreme parallelism afforded by Spark without investing a significant amount of time writing Spark-specific code. The majority of coding time was spent developing science-motivated code/logic to detect, describe, and model dipping events within familiar Python UDFs and using familiar Python libraries. In alternative systems that provide similar levels of parallelism, such as HPC systems based on batch scheduling, a user would typically have to spend significant time altering their science code to conform with the underlying software and hardware that enables their code to scale. For example, they may spend significant time re-writing their code in a way that can be submitted to a batch scheduler like PBS/Slurm, or spend time developing a leader/follower execution model using a distributed computing/communication framework such as OpenMPI. Traditional batch scheduling systems running on shared HPC resources typically have a queue that a user's program must wait in before execution. In contrast, our platform scales on-demand to the needs of each individual user.

This example demonstrates the utility of using cloud computing environments for science: when science is performed on a platform that provide on-demand scaling using tools that can distribute science workloads in a user-friendly manner, time to science is minimized.

4. SCALABILITY, RELIABILITY, COSTS, AND USER EXPERIENCE

Our system is expected to scale both in the number of simultaneous users and to the demands of a single user's analysis. In the former case, JupyterHub and its built in proxy can scale to access by hundreds of users as its workload is limited to routing simple HTTP requests. In the latter case, data queries by individual users are expected to scale to very many machines, allowing for fast querying and transformation of very large datasets. Section 4.1 summarizes tests to verify this claim.

4.1. *Scaling Performance*

We performed scaling tests to understand and quantify the performance of our system. We tested both the "strong scaling" and "weak scaling" aspects of a simple query. Strong scaling indicates how well a query with a fixed data size can be sped up by increasing the number of cores allocated to it. On the other hand, weak scaling indicates how well the query can scale to larger data sizes; it answers the question "can I process twice as much data in the same amount of time if I have twice as many cores?"

Figure 6 shows the strong and weak scaling of a simple query, the sum of the "RA" column of the ZTF dataset, which contains $\sim 3 \times 10^9$ rows, stored in Amazon S3. This dataset is described in more detail in section 3.1. In these experiments, speedup is computed as

$$\text{speedup} = t_{\text{ref}}/t_N \tag{1}$$

where t_{ref} is the time taken to execute the query with a reference number of cores while t_N is the time taken with N cores. For the weak scaling tests, scaled speedup is computed as

$$\text{scaled speedup} = t_{\text{ref}}/t_N \times P_N/P_{\text{ref}} \tag{2}$$

which is scaled by the problem size P_N with respect to the reference problem size P_{ref}. We chose to scale the problem size directly with the number of cores allocated; the 96-core query had to scan the entire dataset, while the 1-core query had to scan only 1/96 of the dataset. Typically, the reference number of cores is 1 (sequential computing), however we noticed anomalous scaling behavior at low numbers of cores, and so we set the reference to 16 in Fig. 6.

In our experiments, we used `m5.large` EC2 instances to host the Spark executor processes, which have 2 vCPU and 8 GiB of RAM allocated to them. The underlying CPU is an Intel Xeon Platinum 8000 series processor. The Spark driver process was started from a Jupyter notebook server running on a `t3.xlarge` EC2 instance with 4 vCPU and 16 GiB of RAM allocated to it. The underlying CPU is an Intel Xeon Platinum 8000 series processor. Single `m5.large` EC2 instances have a network bandwidth of 10 Gbit/s while the `t3.xlarge` instance has a network bandwidth of 1 Gbit/s. Amazon S3 can sustain a bandwidth of up to 25 Gbit/s to individual Amazon EC2 instances. Both the data in S3 and all EC2 instances lie within the same AWS region, us-west-2. The `m5.large` EC2 instances were spread across three "availability zones" (separate AWS data centers): us-west-2a, us-west-2b, and us-west-2c. This configuration of heterogeneous instance types, network speeds, and even separate instance locations represent a typical use-case of cloud computing and offers illuminating insight into performance of this system with these "worst-case" optimization steps.

The weak scaling test showed that scaled speedup scales linearly with the number of cores provisioned for the query; twice the data can be processed in the same amount of time if using twice the number of cores. In other words, for this query, the problem of "big data" is solved simply by using more cores. The strong scaling test showed expected behavior up to vCPU/16 = 5. Speedup increased monotonically with diminishing returns as more cores were added. Speedup dropped from 2.50 with vCPU/16 = 5 to 2.05 with vCPU/16 = 6, indicating no speedup can be gained beyond vCPU/16 = 5. Drops in speedup in a strong scaling test are usually due to real world limitations of the network connecting the distributed computers. As the number of cores increases, the number of simultaneous communications and the amount of data shuffled between the single Spark driver process and the many Spark executor processes increases, potentially reaching the latency and bandwidth limits of the network connecting these computers.

4.2. *Caveats to Scalability*

As mentioned in section 2.4, the use of a shared NFS can limit scalability with respect to the number of simultaneous users. We recommend the administrators of new deployments of our platform consider the access pattern of user data and code on NFS to guarantee scalability to their desired number of users. Carefully designed hybrid models of code and data storage that utilize NFS, EFS, and the Docker image itself (stored on EBS) can be developed that will likely allow the system scale to access from hundreds of users.

4.3. *Reliability*

In general, the system is reliable if individual components (i.e. virtual machines or software applications) fail. Data stored in S3 are in practice 100% durable.[30] Data stored in the EBS volume backing the NFS server are similarly durable, and backed up on a daily basis.

Kubernetes as a scheduling tool is resilient to failures of individual applications. Application failures are resolved by rescheduling the application on the cluster, perhaps on another node, until a success state is reached. When the Kubernetes cluster autoscaler is used, then the cluster becomes resilient to the failure of individual nodes. Pods that are terminated from a node failure will become unschedulable, which will trigger the cluster autoscaler to scale the cluster up to restore the original size of the cluster. For example, if the user's Jupyter notebook server is unexpectedly killed due to the loss of an EC2 instance, it will re-launch on another instance on the cluster, with loss of only the memory contents of the notebook server and the running state of kernels.

[30] AWS quotes "99.999999999% durability of objects over a given year"; https://aws.amazon.com/s3/faqs/

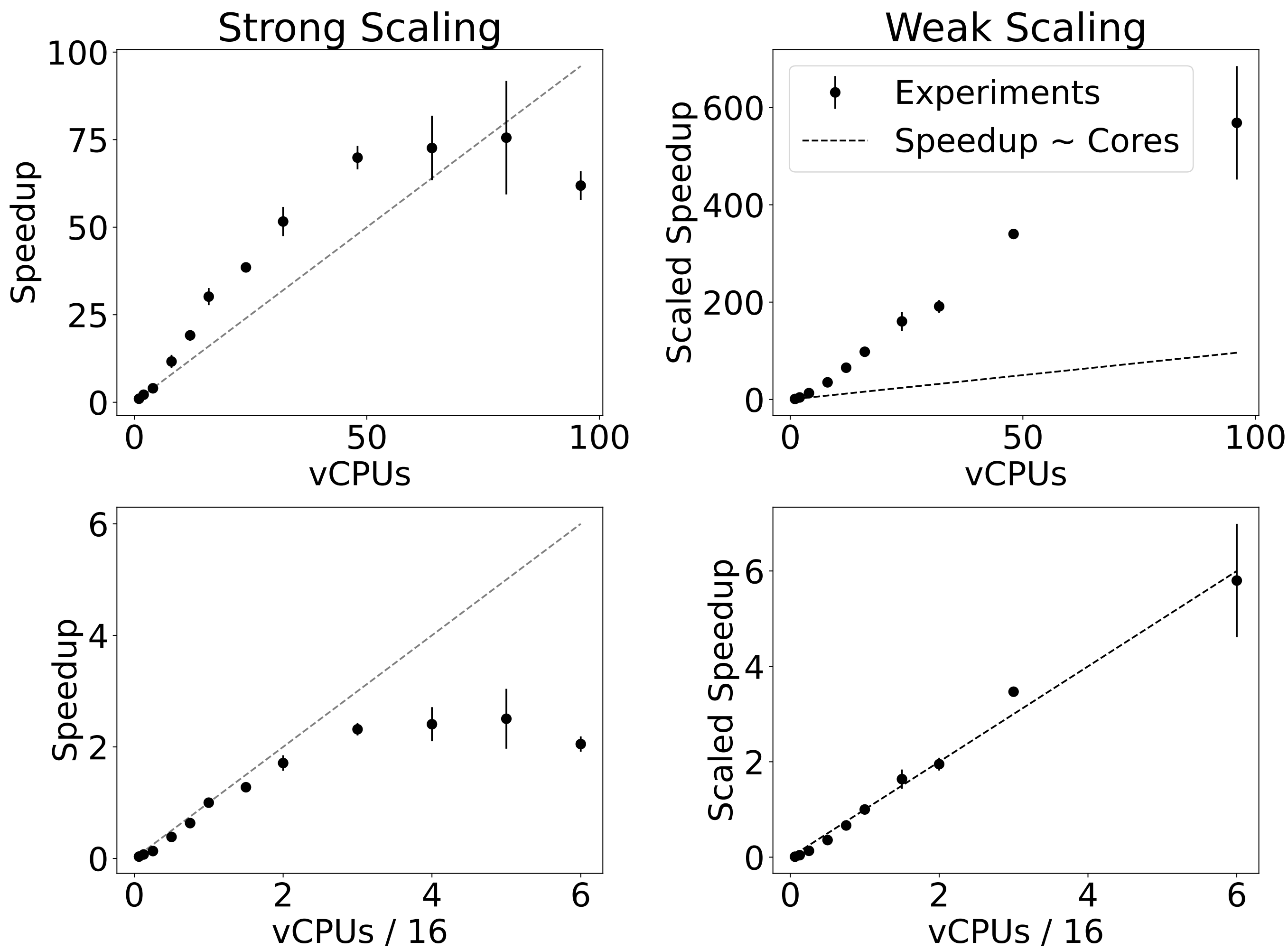

Figure 6. Speedup computed in strong scaling (left) and weak scaling (right) experiments of a simple Spark query that summed a single column of the ZTF dataset, $\sim 3 \times 10^9$ rows. Speedup is computed using Eq. 1 and scaled speedup is computed using Eq. 2. For each value of vCPU, the query was executed several (3+) times. For each trial, the runtime was measured and speedup calculated. Each point represents the mean value of speedup and error bars indicate the standard deviation. The first row shows speedup computed using sequential computing (vCPU = 1) to set the reference time and reference problem size. The second row shows speedup computed using 16 vCPU to set the reference. With sequential computing as the reference, we observe speedup that is abnormally high in both the strong and weak scaling case. By adjusting the reference point to vCPU = 16, we find that we can recover reasonable weak scaling results and expected strong scaling results for a small to medium number of cores. Using the adjusted reference, we observe in the strong scaling case diminishing returns in the speedup as the number of cores allocated to the query increases, as expected. The weak scaling shows optimistic results; the speedup scales linearly with the dataset size as expected.

The same is true of each of the individual JupyterHub and Spark components. Apache Spark is fault-tolerant in its design, meaning a query can continue executing if one or all of the Spark executors are lost and restarted due to loss of the underlying nodes. Similar loss of the driver process (on the Jupyter notebook server) results in the complete loss of the query.

We have run different instances of this platform for approximately three years in support of science workloads at UW, the ZTF collaboration, a number of hackathons, and for the LSST science collaborations. Over that period, we have experienced no loss of data or nodes.

4.4. *Costs*

This section enumerates the costs associated with running this specific science platform. Since cloud computing costs can be variable over time, the costs associated with this science platform are not fixed. In this section, we report costs at the time of manuscript submission as well as general information about resource usage so costs can be recomputed by the reader at a later date.

We describe resource usage along two axes: interactive usage and core hours for data queries. Interactive usage encompasses using a Jupyter notebook server for making plots, running scripts and small simulations, and collaborating with others. Data queries encompass launching a distributed Spark cluster to access and analyze data provided on S3, similarly to the methods described in Sec. 3.3. Equation 3 provides a formula for computing expected monthly costs given the number of users N_u, the cost of each user node C_u, the cost of the Spark cluster nodes C_s, the estimated time spent per week on the system t_u, and the number of node hours used by each user for Spark queries in a month t_s:

$$\begin{aligned}\mathrm{Cost}_{\mathrm{storage}} &= N_u \times 200 \times 0.08 \times (t_u \times (30/7) + t_s)\\ \mathrm{Cost}_{\mathrm{machines}} &= N_u \times (C_u \times t_u \times (30/7) + C_s \times t_s)\\ \mathrm{Cost} &= \mathrm{Cost}_{\mathrm{storage}} + \mathrm{Cost}_{\mathrm{machines}} \end{aligned} \tag{3}$$

Fixed in the equation are constants describing the amount (200 GB) and cost of ($0.08/GB/month) of EBS-backed storage allocated for each virtual machines. Additionally, the term (30/7) converts weekly costs to monthly costs. Node hours can be converted to core hours by multiplying t_s by the number of cores per node.

Table 2 enumerates the fixed costs of the system as well as the variable costs, calculated using Eq. 3, assuming different utilization scenarios, varying the number of users (N_u), the amount interactive usage per week (t_u), and amount of Spark query core hours each month (t_s).

The fixed costs of the system total to $328.51/month, paying for:

1. a small virtual machine, `t3.medium`, for the JupyterHub web application, proxy application, and NFS server ($29.95/month) with 200 GB EBS-backed storage ($16.00/month);
2. two reserved nodes for incoming users at the default virtual machine size of `t3.xlarge` ($119.81/month) with 200 GB EBS-backed storage each ($32.00/month);
3. EBS-backed storage for the NFS server for user files ($8.00/month);
4. and storage of 5,337 GB of catalog data on Amazon S3 ($122.75/month).

Variable costs are harder to estimate, but Table 2 outlines several scenarios to get a sense for the lower/upper limits to costs. 10 scientists using the platform for 4 hours per day 3 days per 7 day week, each using 512 core hours for Spark queries each month (equivalent to 16 hours with a 32 core cluster) adds a cost of $189.32/month. On the other hand, 100 scientists using the platform for 8 hours per day 5 days per 7 day week, each using 2048 core hours for Spark queries each month (64 hours with a 32 core cluster) adds a cost of $6,926.18/month. There are additional costs on the order of ∼$10 that we don't factor into this analysis. Specifically:

1. network communication between virtual machines in different availability zones, introduced when scaling a Spark cluster across availability zones;
2. data transfer costs in the form of S3 GET API requests (data transfer to EC2 virtual machines in the same region is free), introduced in each query executed against the data;
3. and network communication between virtual machines and users over the internet, introduced with each interaction in the Jupyter notebook through the user's web browser.

Each of these costs are minimal, and so we don't include them in our analysis. However, they are worth mentioning because they can scale to become significant. Spark queries requiring GB/TB data shuffling between driver and executors should restrict themselves to a single availability zone to avoid the costs of (1). Costs from (2) are unavoidable, but care should be taken so no S3 requests occur between different AWS regions and between AWS and the internet. Finally, (3) can balloon in size if one allows arbitrary file transfers between Jupyter servers and the user or allows large data outputs to the browser.

The number of core hours for queries is a parameter that will need to be calibrated using information about usage of this type of platform in the real-world. The upper limit guess of 2048 core hours per user per month is roughly equivalent to each user running an analysis similar to that described in Sec. 3.3 each month. By monitoring interactive usage of our own platform and other computation tools, we estimate that realistic usage falls closer to the lower limits we provide.[31]

4.5. *Dynamic Scaling*

Recent versions of Apache Spark provide support for "dynamic allocation" of Spark executors for a Spark

[31] Few users will use the platform continuously in an interactive manner, and even fewer will be frequently executing large queries using Spark.

Virtual Machines

Type	Unit Cost	Amount	Total
Services (`t3.medium`[a])	$0.0416/hour/node	1 node	$29.95/month
Users (`t3.xlarge`)	$0.1664/hour/node	2 nodes + variable	$119.81/month + variable
Spark Clusters (`t3.xlarge` Spot[b])	$0.0499/hour/node	variable	variable

Storage

Type	Unit Cost	Amount	Total
Catalogs (S3[c])	$0.023/GB/month	5,337 GB	$122.75/month
NFS (EBS[d])	$0.08/GB/month	100 GB	$8.00/month
Node Storage (EBS)	$0.08/GB/month/node	200 GB/node	$48.00/month + variable

Fixed Costs

Type	Total
Virtual Machines	$149.76/month
Storage	$178.75/month
All	$328.51/month

Variable Costs

Number of Users	Interactive Usage (hours/week/user)	Spark Query Core Hours (/user/month)	Total
10	12	512	$189.32/month
		2048	$466.27/month
	40	512	$415.67/month
		2048	$692.62/month
100	12	512	$1,893.22/month
		2048	$4,662.71/month
	40	512	$4,156.69/month
		2048	$6,926.18/month

[a] On-Demand pricing in region `us-west-2`: https://aws.amazon.com/ec2/pricing/on-demand/

[b] Spot pricing in region `us-west-2`: https://aws.amazon.com/ec2/spot/pricing/

[c] For the first 50 TB: https://aws.amazon.com/s3/pricing/

[d] General purpose SSD (gp3): https://aws.amazon.com/ebs/pricing/

Table 2. Fixed and variable costs associated with running this analysis platform on Amazon Web Services. This summary provides cost estimates for renting virtual machine and storing data. Additional costs on the order of ∼$10 due to network communication and data transfer are excluded from these results. Reasonable low and high estimates are chosen for the number of active users and the amount of interactive usage they have with the system. The number of Spark query core hours used by each user per month is a guess, but the high end estimate is similar to the core hours used during the analysis in Sec. 3.3.

cluster on Kubernetes.[32] Dynamic allocation allows for the Spark cluster to scale up its size to accommodate long-running queries as well as scale down its size when no queries are running. Figure 7 shows pictorially this scaling process for a long-running query started by a user. This feature is expected to reduce costs associated with running Spark queries since Spark executors are added and removed based on query status, not cluster creation. This means the virtual machines hosting the Spark executor processes will be free more often either

[32] Since Spark version 3.0.0 by utilizing shuffle file tracking on executors as an alternative to an external shuffle file service, which is awaiting support in Kubernetes. See: https://spark.apache.org/docs/latest/configuration.html#dynamic-allocation

to host the Spark executors for another user's query or be removed from the Kubernetes cluster completely.

4.6. *User Experience*

The use of containerized Jupyter notebook servers on a scalable compute resource introduces a few changes to the experience of using a local or remotely hosted Jupyter notebook server. Similar to using a remotely hosted Jupyter notebook, the filesystem exposed to the user has no direct connection to their personal computer, an experience that can be unintuitive to the user. Additionally, file uploads and downloads can only be facilitated through the Jupyter interface, which can be clunky. An SSH server can be started alongside the user's notebook server to allow file transfer using utilities such as `scp` or `rsync`, but this introduces some security risk as public and private keys need to be generated, stored, and managed between the server and the user. SSH access is also a desirable, but unimplemented, feature for users who find the Jupyter notebook environment restrictive or are more comfortable with computing via command line. In future deployments of this system, it is likely that new user interfaces will need to be produced to maximize usability of the filesystem and computing resources while minimizing security risks.

The underlying scalable architecture introduces computing latencies that are noticeable to the user. Virtual machines that host notebook servers and Spark cluster executors are requested from AWS on-demand by the user, and the process of requesting new virtual machines from AWS can take up to $\sim$5 minutes.[33] The user can encounter this latency when logging onto the platform and requesting a server. They also encounter this latency when creating a distributed Spark cluster, as many machines are provisioned on-demand to run Spark executors.

The log-in latency can be mitigated by keeping a small number of virtual machines in reserve so that an incoming user can instantly be assigned to a node. The `zero-to-jupyterhub` Helm chart implements this functionality through its `user-placeholder` option. This functionality schedules placeholder servers on the Kubernetes cluster that will be immediately evicted and replaced when a real user requests a server.

An alternative solution to this would be to place all incoming users on a shared machine, an equivalent to a "log-in node", before moving them to a larger machine at their request. This capability is not built in to the `zero-to-jupyterhub` deployment, but can be integrated with forthcoming Docker container checkpoint-restore functionality. Juric et al. (2021) have integrated such checkpoint-restore functionality for JupyterHub deployments using the Podman container engine, providing a future path for improving the user experience with this technology.[34]

5. CONCLUSIONS AND FUTURE WORK

In this paper, we've described an architecture of a Cloud-based science platform as well as an implementation on AWS that has been tested with data from the Zwicky Transient Facility. The system is shown to computationally scale to and allow parallel analysis with $O(10\text{TB})$ sized tabular, time-series heavy, datasets. It enables science projects that utilizes the ZTF dataset in full, while requiring minimal effort from domain scientists to scale their analysis to the full dataset. The system demonstrates the ability of utilizing elastic computing and I/O capacity of the cloud to enable analyses of large datasets that scale with the number of users.

This work should be viewed in the context of exploration of feasibility of making more astronomical datasets available on cloud platforms, and providing services and platforms – such as the one described here – to combine and analyze them. For any dataset uploaded onto AWS S3 (in the AXS-compatible format) it would be possible to perform cross-dataset analyses with no need to co-locate or pre-stage the data. This enables any dataset provider – whether large or small – to make their data available to the broad community via a simple upload. Second, other organizations can stand up their own services on the Cloud – either use-case specific services or broad platforms such as this-one – to access the data using the same APIs.

This structure also decouples the costs of various elements of the complete platform. The major continuous expense is the cost of keeping the datasets uploaded in the cloud. These costs are manageable, even by small organizations; storing 1 TB of data in S3 costs $\sim$\$25 per month with additional cost scaling with the number of requests for this data. This cost could continue to be borne by the dataset originators or designated curators (i.e., archives).[35] The cost of analysis, however, is kept decoupled: it is the user who controls the number of cores utilized for the analysis, and any additional ephemeral storage used. It is easy to imagine the user –

[33] This time is dependent on the individual cloud provider. DigitalOcean, another cloud provider, can provision virtual machines in $\sim$1.5 minutes based on the experience of the authors.

[34] See https://github.com/dirac-institute/elsa/ and https://podman.io/

[35] "requester-pays" pricing models, supported by some cloud providers, further offloads some of the cost to the user

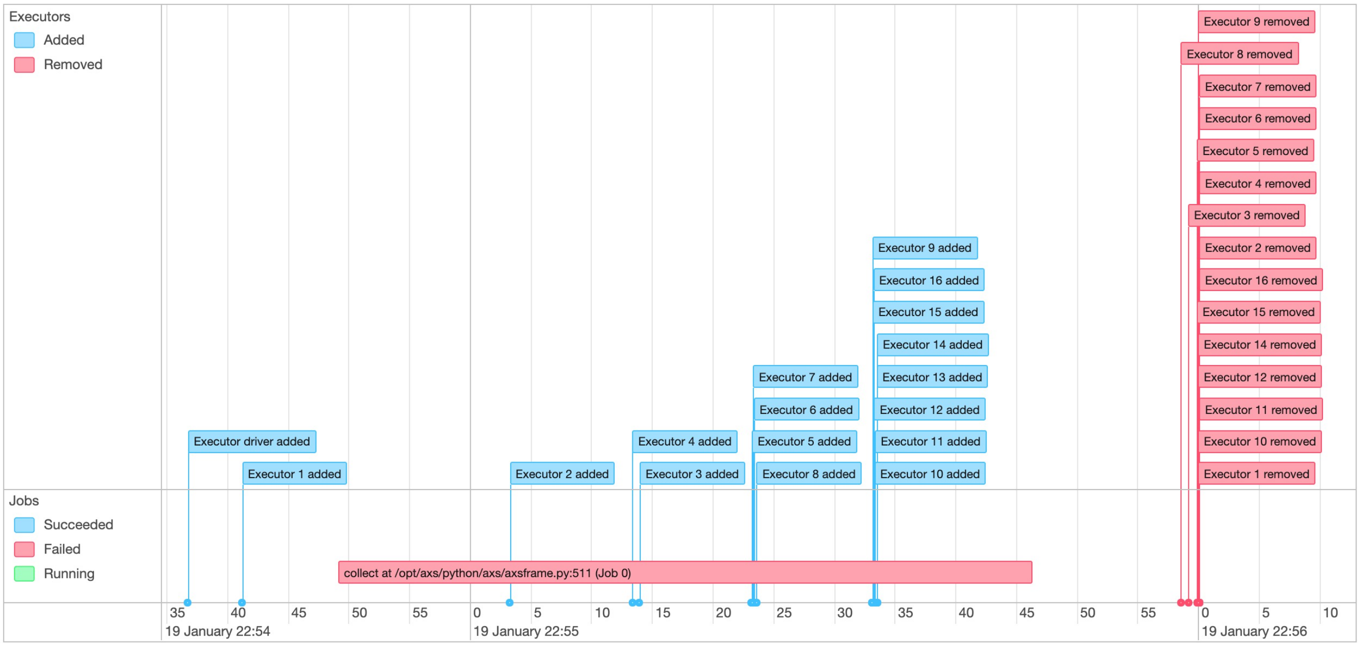

Figure 7. A screenshot of the job timeline from the Spark UI when dynamic allocation is enabled. A long-running query is started, executing with a small number of executors. As the query continues, Spark adds exponentially more executors to the cluster at a user-specified interval until the query completes or the max number of executors is reached. Once the query completes (or is terminated, as shown here), the Spark executors are removed from cluster.

as a part of their grant – being awarded cloud credits for the research, which could be applied towards these costs. Finally, the system provides a direction and an incentive towards continuous improvements of science platforms and associated tools. These are now best viewed as systems utilized by astronomers to enable the exploration of a multitude of datasets available. Their incentive is to maximize science capability while minimizing the cost to the user, who now has the ability to "shop around" with their credits for a system most responsive to their needs. The utilization, strengths, and weaknesses of the ecosystem become easier to measure.

We are planning future work to continue to improve cost-effectiveness of this model of computing and data access. Forthcoming container checkpoint/restore functionality integrated into JupyterHub will allow for frequent culling of unused Jupyter notebook servers running on this platform without impacting user experience. In addition, as the user-base expands for these types of science platforms, new tools will be developed to support using cloud resources for custom science workflows supported by legacy code.

This material is based upon work supported by the National Science Foundation under Grant No. AST-2003196.

The authors acknowledge the support from the University of Washington College of Arts and Sciences, Department of Astronomy, and the DiRAC Institute. The DiRAC Institute is supported through generous gifts from the Charles and Lisa Simonyi Fund for Arts and Sciences and the Washington Research Foundation. M. Jurić wishes to acknowledge the support of the Washington Research Foundation Data Science Term Chair fund, and the University of Washington Provost's Initiative in Data-Intensive Discovery.

Based on observations obtained with the Samuel Oschin Telescope 48-inch and the 60-inch Telescope at the Palomar Observatory as part of the Zwicky Transient Facility project. ZTF is supported by the National Science Foundation under Grant No. AST-1440341 and a collaboration including Caltech, IPAC, the Weizmann Institute for Science, the Oskar Klein Center at Stockholm University, the University of Maryland, the University of Washington, Deutsches Elektronen-Synchrotron and Humboldt University, Los Alamos National Laboratories, the TANGO Consortium of Taiwan, the University of Wisconsin at Milwaukee, and Lawrence Berkeley National Laboratories. Operations are conducted by COO, IPAC, and UW.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0019323.

S. Stetzler thanks Juan CV. Barbosa, Hannah V. Bish,

REFERENCES

Bellm, E. C., Kulkarni, S. R., Graham, M. J., et al. 2019, Publications of the Astronomical Society of Pacific, 131, 018002, doi: 10.1088/1538-3873/aaecbe

Boyajian, T. S., LaCourse, D. M., Rappaport, S. A., et al. 2016, Monthly Notices of the Royal Astronomical Society, 457, 3988, doi: 10.1093/mnras/stw218

Cheng, Y., Liu, F. C., Jing, S., Xu, W., & Chau, D. H. 2018, in Proceedings of the Practice and Experience on Advanced Research Computing, PEARC '18 (New York, NY, USA: Association for Computing Machinery), doi: 10.1145/3219104.3229288

Dark Energy Survey Collaboration, Abbott, T., Abdalla, F. B., et al. 2016, MNRAS, 460, 1270, doi: 10.1093/mnras/stw641

Dekany, R., Smith, R. M., Riddle, R., et al. 2020, PASP, 132, 038001, doi: 10.1088/1538-3873/ab4ca2

Dubois-Felsmann, G., Lim, K.-T., Wu, X., et al. 2017

Gaia Collaboration, Prusti, T., de Bruijne, J. H. J., et al. 2016, Astronomy and Astrophysics, 595, A1, doi: 10.1051/0004-6361/201629272

Graham, M. J., Kulkarni, S. R., Bellm, E. C., et al. 2019, PASP, 131, 078001, doi: 10.1088/1538-3873/ab006c

Ivezić, Ž., Kahn, S. M., Tyson, J. A., et al. 2019, ApJ, 873, 111, doi: 10.3847/1538-4357/ab042c

Jurić, M., Ciardi, D., & Dubois-Felsmann, G. 2017

Juric, M., Stetzler, S., & Slater, C. T. 2021, Checkpoint, Restore, and Live Migration for Science Platforms. https://arxiv.org/abs/2101.05782

Kaiser, N., Burgett, W., Chambers, K., et al. 2010, in Proc. SPIE, Vol. 7733, Ground-based and Airborne Telescopes III, 77330E, doi: 10.1117/12.859188

Kluyver, T., Ragan-Kelley, B., Pérez, F., et al. 2016, in Positioning and Power in Academic Publishing: Players, Agents and Agendas, ed. F. Loizides & B. Scmidt (Netherlands: IOS Press), 87–90. https://eprints.soton.ac.uk/403913/

Masci, F. J., Laher, R. R., Rusholme, B., et al. 2019, PASP, 131, 018003, doi: 10.1088/1538-3873/aae8ac

Patterson, M. T., Bellm, E. C., Rusholme, B., et al. 2018, Publications of the Astronomical Society of the Pacific, 131, 018001, doi: 10.1088/1538-3873/aae904

Scaramella, R., Mellier, Y., Amiaux, J., et al. 2014, in IAU Symposium, Vol. 306, Statistical Challenges in 21st Century Cosmology, ed. A. Heavens, J.-L. Starck, & A. Krone-Martins, 375–378, doi: 10.1017/S1743921314011089

Shappee, B., Prieto, J., Stanek, K. Z., et al. 2014, in American Astronomical Society Meeting Abstracts, Vol. 223, American Astronomical Society Meeting Abstracts #223, 236.03

Spergel, D., Gehrels, N., Baltay, C., et al. 2015, arXiv e-prints, arXiv:1503.03757. https://arxiv.org/abs/1503.03757

Tonry, J. L., Denneau, L., Heinze, A. N., et al. 2018, Publications of the Astronomical Society of Pacific, 130, 064505, doi: 10.1088/1538-3873/aabadf

Wang, D. L., Monkewitz, S. M., Lim, K.-T., & Becla, J. 2011, in SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 1–11, doi: 10.1145/2063348.2063364

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. 2010, in Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10 (USA: USENIX Association), 10. https://dl.acm.org/doi/10.5555/1863103.1863113

Zečević, P., Slater, C. T., Jurić, M., et al. 2019, Astronomical Journal, 158, 37, doi: 10.3847/1538-3881/ab2384