Draft version December 8, 2020
Typeset using LaTeX **twocolumn** style in AASTeX62

A Scalable Cloud-Based Analysis Platform for the Zwicky Transient Facility

Steven Stetzler,[1] Mario Jurić,[1] Colin T. Slater,[1] Petar Zečević,[2,3] and others[1]

[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

Research in astronomy is undergoing a major paradigm shift, transformed by the advent of large, automated, sky-surveys into a data-rich field where multi-TB to PB-sized spatio-temporal data sets are commonplace. For example the Legacy Survey of Space and Time (LSST) is about to begin delivering observations of $> 10^{10}$ objects, including a database with $> 4 \times 10^{13}$ rows of time series data. This volume presents a challenge: how should a domain-scientist with little experience in data management or distributed computing access data and perform analyses at PB-scale?

We present a solution to this problem built on adapted industry standard tools and made accessible through web gateways. Specifically, we have developed a deployment of Astronomy eXtensions for Spark (AXS) framework on AWS (with Kubernetes and S3 as the orchestration and storage layers, respectivelly), with auto-scaling configurations requiring no end-user interaction. This system is accessed through JupyterHub, a web-accessible front-end to a Jupyter notebook server that includes a rich library of pre-installed common astronomical software (accessible at http://hub.dirac.institute).

We use this system to enable the analysis of data from the Zwicky Transient Facility, presently the closest precursor survey to the LSST, and discuss initial results. To our knowledge, this is a first application of cloud-based scalable analytics to astronomical datasets approaching LSST-scale. The code is available at https://github.com/astronomy-commons.

Keywords: science platforms, astronomical data analysis, ZTF

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.

Two years ago, the Zwicky Transient Facility (ZTF; Bellm et al. 2019) began 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. 2019). Towards the middle of 2022, 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, 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

Corresponding author: Steven Stetzler
stevengs@uw.edu

($\lesssim$\$1M scale), contributing billions of more specialized measurements.

However, 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.

Beyond the shear input size, the *nature* of astronomical investigations itself is changing. In the first two decades of survey-driven science, archived data has often been seen as a shortcut to observations: instead of requesting telescope time, one could "observe" a region of sky with a Structured Query Language (SQL) query. Thus, the typically requested and analyzed data subsets were relatively small[1]. The next decade, however, is expected to be weighted towards studies examining the *whole dataset*: performing large-scale classification (including using machine learning techniques), clustering analyses, searching for exceptional outliers, or measuring faint statistical signals (e.g., see LSST Science Collaboration et al. 2009). The change is driven by the needs of the big scientific questions of the day. For some — such as the nature of dark energy — we are reaching the limits of measurement precision or the numbers of objects that can be observed: utilization of all available data and improved statistical treatments are the only way to an answer.

The present analysis workflow paradigm follows what we call a "Subset-Download-Analyze" pattern. The data is maintained in archives, and (typically) the organization that has produced the data set makes it public in a highly available database that an astronomer can inspect using query languages. For example, an astronomer can query the SDSS catalog through the `SkyServer` service[2] using SQL. The query is sent by the user through a web client (a web browser like Google Chrome, Firefox, etc.) to a remote server that executes the query. This remote server is another computer, likely a more powerful computer that can access and process the data much faster than the user's computer likely could. This remote machine is also likely able to preform several queries in parallel, serving up data to several clients at once. This is the "Subset" portion of the workflow, as the user is requesting a *subset* of the whole data set be returned to them. The result of the query are returned to the user through the web browser in the requested format, likely a downloadable file in a `csv` or `fits` format. This is the "Download" portion of the workflow. Finally, the user takes this downloaded data and performs anayisis, typically on computational resources completely separate from where the data are stored (the "Analyze" portion of the workflow). Figure 1 shows pictorially the typical computing architecture that supports this data access/analysis pattern.

There are several bottlenecks here. First, if the downstream science analysis changes and requires a new subset of the data, one must return to step (1) of this process. An astronomer must re-query the remote database and download the data again. The process may be automated if an archive exposes an Application Programming Interface (API) that allows queries to be executed programatically, but not all archives expose such interfaces. If an astronomer wants to combine multiple datasets, this process may become even more difficult: multiple APIs may be needed, or in the worst case, manual visits to several independent portals with different web interfaces.

The second bottleneck comes from downloading the data. In the current paradigm, the download speed for a dataset is limited by the lowest bandwidth connection in the network between a user's computer and the remote machine. Typically, this bottleneck is introduced at the connection of the user's computer to the internet. Typical connections in US academic institutions range from $\sim 1-10$ MB/s. For a subset of 1 GB in size (which will be *small* in the LSST era), this introduces a time-to-science latency of $\sim 1.5-15$ minutes. Such latencies make exploratory science time-consuming.

The final bottleneck comes in the analysis stage. Given the user has the data they need, they are still limited by the computational resources they have on hand to analyze it. These may range from a laptop with $2-4$ CPU cores and $8-16$ GB of RAM, to university- or nationally- funded HPC infrastructure. However, the access to such infrastructure is usually limited to large institutions with significant funding. Secondly, it requires significant end-user know-how: converting code for personal analysis to run at scale on an HPC cluster is highly non-trivial for a domain-expert, and may take

[1] Past studies show frequencies of Sloan Digital Sky Survey queries follow a f^{-1} power law (O'Mullane et al. 2005; Kurtz & Bollen 2010).

[2] http://skyserver.sdss.org/

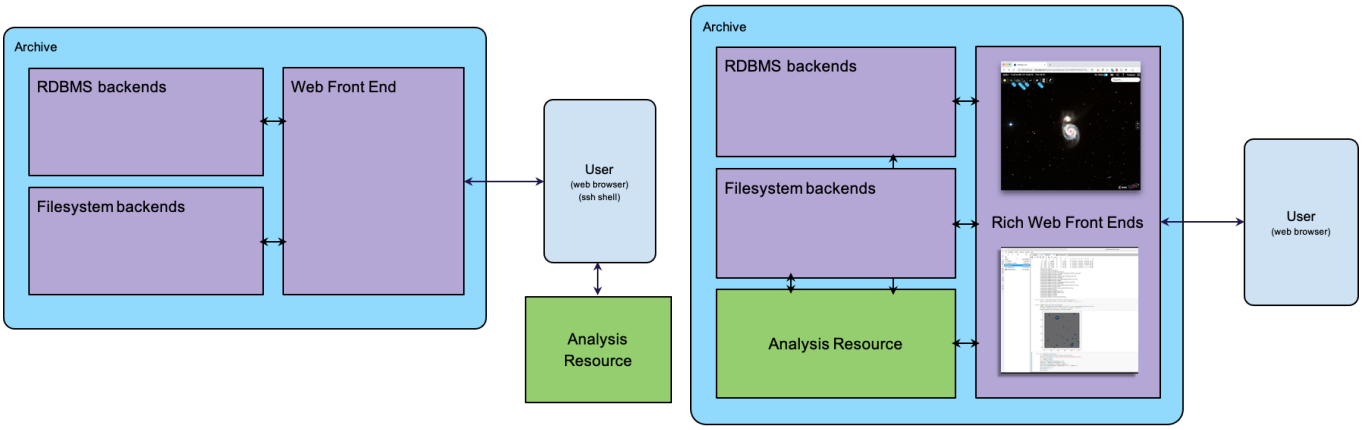

Figure 1: The typical computing architecture of present day analysis systems (left) compared with proposed future architectures (right) that motivate this work. Left: The user interfaces with a web front end provided by an archive to query a database (Relational Database Management System; RDBMS). Data are transferred by the user to analysis resources that are not co-located with the data. Bottlenecks are introduced that severely limit scalability between the user and the web front end and between the web front end and database/filesystem backends. Right: Analysis resources are moved close to the data and analysis/visualization tools are exposed to the user through a rich web front end.

anywhere from days to weeks.

For all these reasons, the traditional "subset-download-analyze" paradigm in place today will be difficult to apply to multi-TB to PB-scale datasets. In this new environment, how does a domain expert access the data of interest and write and run analyses? How do they write a code simultaneously fitting billion+ stars for distances and the interstellar dust distribution? What is an astronomer to use to orchestrate running a light-curve classifier using state-of-the-art machine learning methods that robustly handles a trillion data points?

A popular solution today 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 or Zeppelin). 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[3] and implemented[4] a science platform suitable for their use cases based on the ability to do all work remotely through a web-browser. While such science platforms are a major step forward in working with large datasets, they're not a complete solution. For example, they do not solve the issue of having sufficient computing next to the data: all users of shared HPC resources are familiar with "waiting in the queue" due to oversubscription. Secondly, while the computing is now closer to the data, the analysis is still not parallel – the query results are typically still sent to one place, rather than directly to the user's code for analysis. This severely limits the scalability. Finally, 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; Apache Spark 2018) engine, to work

[3] http://ls.st/LSE-319
[4] https://nb.lsst.io/

in this cloud environment. Spark allows the execution of everything from simple ANSI SQL-2011 compliant queries, to complex distributed workflows, all driven from Python. Finally, we build a JupyterHub facade as the entry-point to the system.

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

In the following sections, we introduce 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*

Cloud computing has represented a paradigm shift in the acquisition and maintenance of computing infrastructure to support software applications. Large software companies[5], in support of their own computing needs, have invested billions of dollars into building data centers around the world to serve an online experience to their users. This investment in centralized computing infrastructure has offered another commercial application: other businesses and individuals were willing to pay for the ability to both store large quantities of data in these data centers and rent computing hardware to host their own software applications. Traditional acquisition and maintenance of computing infrastructure, so-called "on-premise" computing, has fallen out of style, as it's become cheaper and easier to rent machines and store data with a cloud provider. To attract and retain customers, cloud providers have also created ecosystems of managed services that make managing computing infrastructure easier. For example, one can use the web interface of a cloud provider to create a scalable database ready to serve terabytes of data with ease.[6]

Cloud services provide a new way of working with computers. Web interfaces and APIs allow a user to provision "virtual machines" from the cloud provider, networked through virtual networks, backed by virtual storage devices. These virtual machines represent "slices" of physical hardware allocated to the user, isolated from other virtual machines running on the same hardware. These virtual machines are networked through virtual networks that can remain private or can be connected to the internet. Storage for virtual machines are typically backed by virtual storage devices, slices of physical storage connected over a network.[7] In addition, since each of these components are virtual and not tied to an investment in physical hardware, they are rented to the user at a price that varies with 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 simple GET/PUT API calls. S3 supports very high throughput at the terabit-per-second assuming storage access patterns are optimized.[8] 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.

One 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" that occurs when software applications become too strongly coupled with

[5] in the United States, Google, Microsoft, and Amazon

[6] The Amazon DynamoDB and RDS managed services are examples of this.

[7] Amazon EBS is an example. Some cloud providers offer virtual machines with directly connected storage devices.

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

the cloud provider's API. The open source community has developed orchestration tools, like Kubernetes, to alleviate this pain point. Kubernetes is used to schedule software applications as Docker containers[9] on a cluster of computers while handling requests for and provisioning of cloud resources to support running those containers. Kubernetes provides a cloud-agnostic API for provisioning generic cluster resources such as Nodes (virtual machines), Persistent Volumes (virtual storage devices in a variety of formats), and Load Balancers (URL endpoints to a virtual machine in a virtual network) in any supported cloud environment. Each Kubernetes object is described using YAML, a human-readable format for storing data (lists and dictionaries of strings and numbers). Figure 2 shows an example set of Kubernetes objects in YAML that together would link a Jupyter notebook server backed by a 10 GiB storage device to an external URL.

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 2.2, we discuss how we leverage cloud infrastructure to build such a platform.

2.2. *System Architecture*

Underlying this platform are three key components:

1. An interface for computing. We use the Jupyter ecosystem, a JupyterHub deployment based on the `zero-to-jupyterhub` project that spawns Jupyter notebook servers 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.

2. A scalable analytics engine. We use Apache Spark, an industry standard tool for distributed data querying and analysis.

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.

[9] 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.

Network

```
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

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

Application

```
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 2: An illustration of the structure and composition of YAML files for creating Kubernetes objects. Colored text indicate how the files are linked to support one another. This example shows how a Jupyter notebook application can be created as a Pod on the cluster (right). Additional networking objects (top left) specify how a public URL (`LoadBalancer`) can be connected to the Jupyter notebook Pod (`notebook-pod`) on a certain port (`8888`). This example also shows how storage can be allocated for the notebook (bottom left). A `PersistentVolumeClaim` is created requesting 10 GB of hard drive space from the cloud provider. The Jupyter notebook Pod consumes this volume claim and mounts it within the container's file system.

Each of these components are largely disconnected from one another and can be mixed and matched with other drop-in solutions.[10] At a low-level, each of these components are comprised of simple processes communicating with each other through 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

[10] 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.

container, or using a managed cloud service as long as each component is properly networked.

The Jupyter notebook application, and its extension Jupyter lab, provide an ideal environment for astronomers to access, manipulate, and visualize both large and small 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 securely through a JupyterHub, a web application that securely forwards authenticated requests from a central URL to a notebook server.[11] 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`[12] and `ZwickyTransientFacility`[13] GitHub organizations.

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. Spark starts by formulating a plan for executing a query, and splits this plan into several (1 to thousands) of individual independent tasks. The driver process keeps track of the state of the query while communicating with separate executor processes that execute the individual tasks and return their results. 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[14], or be distributed across several other machine in a distributed computing context. 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.

Researchers at the DiRAC institute 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.

Amazon S3 is a scalable object store with built-in backups and optional replication across AWS regions (for more scalable access). 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[15]. Additionally, there is no limit to the amount of data that can be stored.[16] Data are stored in Apache Parquet format (Parquet Project 2018), a compressed column-oriented data storage format optimized for very fast reads of large tables. The columnar nature and partitioning of the files means that one can obtain a subset of just the "RA" column of an astronomy catalog without scanning through all of the files.

The JupyterHub, Jupyter notebook servers, and Spark executors are deployed as Pods on a Kubernetes cluster. We use the managed Amazon Elastic Kubernetes Service (EKS) for provisioning a fully-managed Kubernetes master node. Independent of EKS, we have a set of managed virtual machines provisioned through Amazon Elastic Compute Cloud (EC2) on which Kubernetes can schedule containers. The Kubernetes Cluster Autoscaler[17], an optional add-on to Kubernetes, automatically scales the cluster of VMs up when there are Docker containers that cannot be scheduled without more compute resources and scales the cluster down when there are VMs that are under-utilized.

Figure 3 shows the state of our Kubernetes cluster during normal usage of the platform, and 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

[11] As an example, one may access a JupyterHub at the URL https://hub.com which, if you are an authenticated user, will forward through a proxy to https://hub.com/user/⟨username⟩. When running on 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.

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

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

[14] This isn't done in practice; instead, Spark supports spawning threads in the driver process that replace the external executor process(es) when using local resources

[15] There are projects that provide an interface layer between a client and S3 to make the file system nearly completely POSIX compliant.

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

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

the JupyterHub. This notebook server is able to be run on any of the virtual machines rented from the cloud provider, as long as these machines are networked to the JupyterHub server and the Kubernetes cluster. 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.

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 large charts from smaller ones. We have created a Helm chart to describe our platform based off of the `zero-to-jupyterhub` Helm chart[18], a basic and customizable installation of JupyterHub on Kubernetes. The `zero-to-jupyterhub` chart uses Docker images from the Jupyter Docker Stacks[19] by default and uses the `KubeSpawner`[20] for creating Jupyter notebook servers using the Kubernetes API directly instead of using Helm.

The Jupyter notebook servers have Spark installed with Python bindings along with AXS that is configured to both access data stored in S3 and have permissions to create workloads on the Kubernetes cluster. When creating a Spark cluster to query and analyze data, the user has the choice to use the resources of their notebook server (a local Spark cluster) or use the underlying Kubernetes cluster to use additional, potentially larger, compute resources (a distributed Spark cluster). Figure 4 shows the process trees in the single user server and on distributed machines to create the Spark cluster. In either case, computation is performed close to the data as data sets are stored on S3 in the same region as the notebook servers and Spark executors. When the user wishes to scale their analyses to more computers, they only need to switch their Spark cluster to use the Kubernetes API and include a few extra configurations. When requests are made for more computing resources, the Kubernetes cluster has the capability to automatically rent more virtual machines from the cloud provider to accommodate the increased workload.

[18] https://zero-to-jupyterhub.readthedocs.io/
[19] https://jupyter-docker-stacks.readthedocs.io/
[20] https://jupyterhub-kubespawner.readthedocs.io/

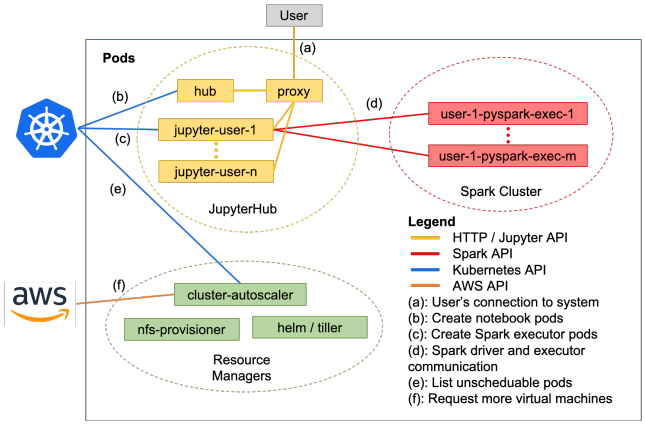

Figure 3: A diagram of the essential components of the Kubernetes cluster when the science platform is in use. Each box represents a single Kubernetes Pod scheduled on the cluster. The colors of the boxes and the dashed ovals surrounding the three groups are for visualization purposes only; each Pod 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). In the background, the Kubernetes cluster autoscaler keeps track of the scheduling status of all Pods (e). 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 (f).

2.3. *Alterations to JupyterHub*

We altered several components of the base `zero-to-jupyterhub` deployment described in Section 2.2 to fit our needs. First, we needed to customize the software available in the single user server Docker image. We built upon the `base-notebook` Docker image from the Jupyter Docker stacks to add desired Python packages such as `numpy`, `scipy`, `astropy`, and `astroquery` (Harris et al. 2020; Virtanen et al. 2020; Astropy Collaboration et al. 2013; Ginsburg et al. 2019) in addition to AXS and Apache Spark. The Helm chart was modified to reference this new Docker image when creating single user servers.

Second, we needed 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 de-

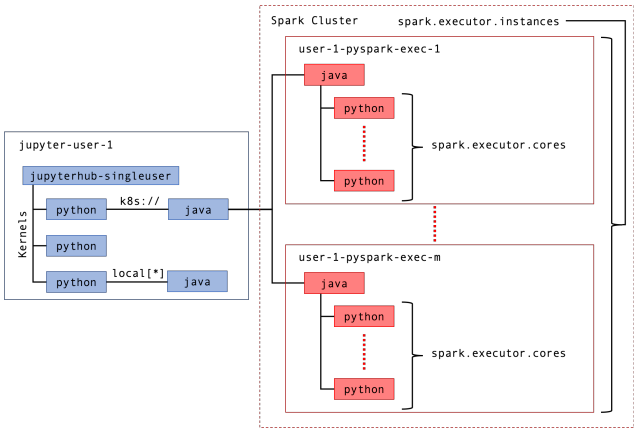

Figure 4: An illustration of the process trees in several of the containers that compose a Spark Cluster on Kubernetes. The user's notebook pod (left; `jovyan-user-1`) contains several python processes that execute user code as normal. A local Spark cluster can be created in a notebook by specifying `spark.master=local[*]`. This creates a Java process on the same machine utilizing all cores available. This Java process can spawn Python processes for parallel execution of Python code. A distributed Spark Cluster can be created by connecting Spark to the Kubernetes API by specifying `spark.master=k8s://<api-endpoint-url>`. This creates a number of Kubernetes Pods that represent Spark executors depending on the value of the `spark.executor.instances` configuration. Within each Pod, a Java process is created that spawns Python processes for parallel execution of Python code depending on the value of the `spark.executor.cores` configuration.

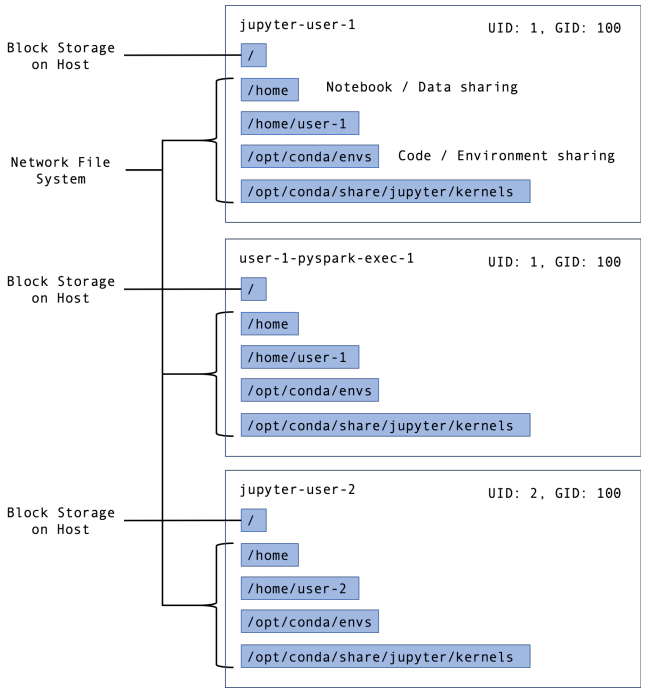

Figure 5: An illustration of the filesystem within each container Spawned by the JupyterHub (`jupyter-user-1` and `jupyter-user-2`) and by the user in the creation of a distributed Spark cluster. Most of the filesystem exists (the root directory `/`) on an ephemeral storage device tied to the host machine. 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. We are careful to set UNIX user id (`UID`) and group id (`GID`) parameters correctly in each container so that users can only access their own data.

vice configured for the Kubernetes cluster[21] for each single user server, allowing a user's files to persist beyond the lifetime of their server. By default, this is mounted at the location `/home/jovyan` in the single user container. Instead, we provisioned a network file system (NFS) server using a community supported Helm chart, the `nfs-server-provisioner`[22] to create a centralized location for user files and enable file sharing between servers. Figure 5 shows how the NFS server is mounted into single user pods. 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. The user directory is created with correct permission when the server container starts using a Docker "entrypoint" script. 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) assigned to each user corresponds to their unique GitHub user id, pulled using GitHub's API during authentication after approval from the user. This presents a complication if GitHub updates their method of assigning user ids or changes their API, leading to possible disruption of service, but in a more production-ready en-

[21] for AWS, this an an Elastic Block Store (EBS) volume

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

vironment, a more robust method of authentication and identity assignment could be used[23].

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 NFS server that we manage ourselves.

In addition to storing home directories on the NFS server, we have explored storing entire code bases (`conda` environments) on the NFS server to be mounted across all containers. This allows for real-time updating of installed software without the need to re-start user servers. This has the downside of decreased scalability. Code bases are often made up of thousands of small files, and a request for each file when starting a notebook leads to large load on the NFS server. This load increases when serving more than one client, and may not be a scalable solution for serving hundreds of users. Instead, we have adopted 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. 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.

Finally, we wanted to provide each user with additional flexibility in their computing resources, allowing them to move their notebook server to a machine with more CPU/RAM if necessary. To do this, we created a custom JupyterHub options form that is shown to the user when they try to start their server, shown in Fig. 6. 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 CPU. 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. A modest choice with 4 vCPU and 16 GiB of memory at a cost of $0.17/hour (the `t3.xlarge` EC2 instance) is chosen by default.

[23] JupyterHub provides an authenticator class that is compliant with the LDAP protocol for completely custom authentication methods

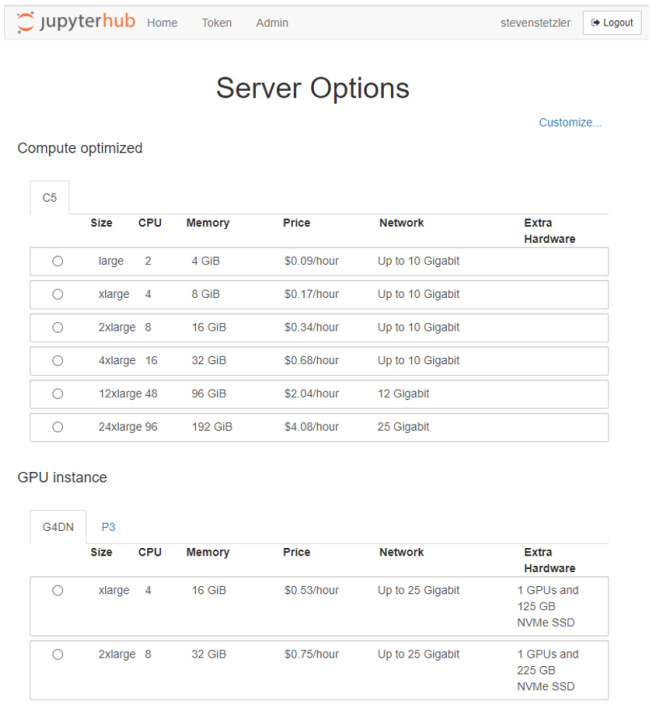

Figure 6: A screenshot of the JupyterHub server spawn page. Several options for computing hardware are presented to the user with their hardware and costs enumerated. Several options are hidden in this screenshot. Of note is the ability to spawn GPU instances on demand. When a user selects one of these options, their spawned Kubernetes Pod is tagged so that it can only be scheduled on a node of the requested type. Individual nodes are tagged by their type by default in Kubernettes. If a certain node type does not exist in the Kubernetes cluster, the cluster autoscaler will provision it from the cloud provider (introducing a ~ 5 minute spawn time).

2.4. *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 a security risk as public and private keys need to be generated

and managed between the server and the user. In a production-ready deployment environment, it is likely that new user interfaces will need to be produced to maximize usability of the filesystem 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 ~ 5 minutes.[24] The user encounters this latency when logging onto the platform and requesting a server. They also run into this latency when creating a distributed Spark cluster, as machines are provisioned 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.

2.5. *Scaling Performance*

We performed scaling tests to demonstrate the performance of our system. We performed tests of both "strong scaling" and "weak scaling" 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 7 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

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

Scaled speedup is scaled by the problem size with respect to the reference problem size. 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 our plots.

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.

2.6. *Reliability*

In general, the system is reliable if individual components (i.e. virtual machines and drives) fail. Data stored in S3 are in practice 100% durable.[25] Data stored in the EBS volume backing the NFS server are similarly durable.

Kubernetes as a scheduling tool is resilient to failures of individual nodes and applications. Application failures are resolved by rescheduling the application on the cluster, perhaps on another node, until a success state is reached. Evicted pods from a node failure will trigger

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

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

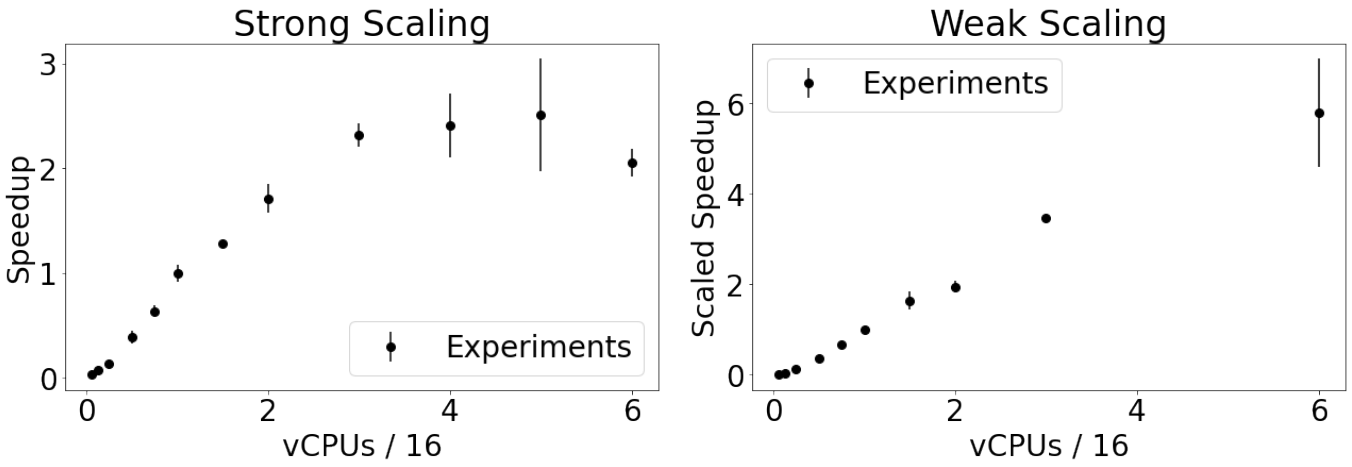

Figure 7: Speedup computed in strong scaling (left) and weak scaling (right) experiments of a simple Spark query that summed a single column of the ZTF dataset, $\sim 3 \times 10^9$ rows. Speedup is computed using Eq. 1 and scaled speedup is computed using Eq. 2. In the strong scaling case, we observe 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 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. 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. To date, we have experienced no loss of data or nodes during the usage of our platform.

3. ZTF SCIENCE PLATFORM

We've built a science platform meant to enable the analysis of data from the Zwicky Transient Facility (ZTF). Section 3.1 describes the datasets available on this platform, Section 3.2 demonstrates the typical access pattern to the data using the AXS API, and Section 3.3 showcases a science project enabled by this platform.

3.1. *Datasets available*

Table 1 enumerates the datasets available to the user. We provide to the user de-duplicated ZTF match files for analysis. The most recent version of these match files have a data volume of $\sim$ 4 TB describing light curves of $\sim$ 3 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.

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	3.3
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.

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_i')
```

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 returns the square of a number can be mapped onto all $\sim$ 3 billion i-band magnitude measurements in the ZTF dataset (in parallel):

```
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
def square(x):
    return x**2
square_udf = udf(square, FloatType())
mag_i_squared = square_udf(ztf['mag_i']).
    ↪ collect()
```

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 we will fully describe in an upcoming paper; 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, we created a set of User-Defined Functions (UDFs) for model fitting that wraps 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 8 shows an outline of this science process using AXS.

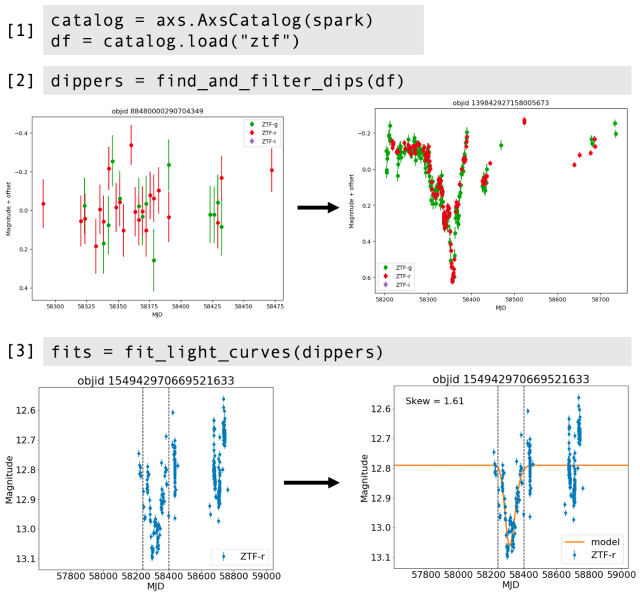

Figure 8: An example analysis (boiled down to two lines) that finds light curves in the ZTF dataset with a dimming event. (1) shows how the ZTF dataset is loaded as a Spark DataFrame (df), (2) shows the product of filtering light curves for dimming events, and (3) shows the result of fitting a model 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.

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 $\sim$ 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 $\sim$ $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 $\sim$ 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. CONCLUSIONS AND FUTURE WORK

In this paper, we've described an architecture of a Cloud-based science platform and successfully implemented a version of the platform using Amazon Web Services for analysis of 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-domain, datasets. This platform has enabled 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 is now possible to perform cross-dataset analyses through the ZTF science platform 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).[26] 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 – 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.

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.

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

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. Major funding has been provided by the U.S National Science Foundation under Grant No. AST-1440341 and by the ZTF partner institutions: the California Institute of Technology, the Oskar Klein Centre, the Weizmann Institute of Science, the University of Maryland, the University of Washington, Deutsches Elektronen-Synchrotron, the University of Wisconsin-Milwaukee, and the TANGO Program of the University System of Taiwan.

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.

REFERENCES

Apache Spark. 2018, Apache Spark Website. http://spark.apache.com

Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., et al. 2013, A&A, 558, A33, doi: 10.1051/0004-6361/201322068

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

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

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

Ginsburg, A., Sipőcz, B. M., Brasseur, C. E., et al. 2019, The Astronomical Journal, 157, 98, doi: 10.3847/1538-3881/aafc33

Harris, C. R., Millman, K. J., van der Walt, S. J., et al. 2020, Nature, 585, 357, doi: 10.1038/s41586-020-2649-2

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

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

Kurtz, M. J., & Bollen, J. 2010, Annual Review of Information Science and Technology, 44, 3, doi: 10.1002/aris.2010.1440440108

LSST Science Collaboration, Abell, P. A., Allison, J., et al. 2009, ArXiv e-prints. https://arxiv.org/abs/0912.0201

O'Mullane, W., Li, N., Nieto-Santisteban, M., et al. 2005, eprint arXiv:cs/0502072

Parquet Project. 2018. http://parquet.apache.org/documentation/latest/

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

Virtanen, P., Gommers, R., Oliphant, T. E., et al. 2020, Nature Methods, 17, 261, doi: 10.1038/s41592-019-0686-2

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