Bioinformatics Solutions on AWS

By
Jillian Rowe

INTRODUCTION:

CHAPTER

ONE

INTRODUCTION

1.1 What this Book Is and Isn't

This book is meant to give a bird's eye view to Bioinformaticians and other Data Scientists who are curious about the world of cloud computing. Maybe you have been on an in house HPC and are interesting in bursting to the cloud, or maybe you've founded a startup that needs infrastructure.

This book will give you an idea of the services you need to start from the ground up, from storage and compute infrastructure, to what types of clusters to use for different analysis types. I hope this books gives you the resources you need to design and implement infrastructure on AWS that suits your needs. Of course, I can't go over every single service offered by AWS. There are way too many of those, and I would never claim to be an expert in each. Instead, this book covers the service options I use most frequently, and lists alternatives that I am aware of.

This book *is not* a tutorial on how to specifically deploy any of the services I discuss here. Meaning, there is no code until we get to the case studies. *No code*. I tried writing code for deployment, and was out of date before I even finished the book. Incidentally, this lead to my spending entirely too much time pondering my career choices. Instead, at the end of each service I will link to several resources I have found extremely reputable for deploying and managing said service. You can find a long list of websites and resources at the end of the book in the References.

If you would like more technical tutorials I have quite a bit of those on my website. I also have a help desk, which is mostly for my clients, but there is quite a bit of content on there as well.

This book is also self edited and self published. There are certainly typos and probably mistakes. If you feel that I haven't cited something or cited something incorrectly please shoot me an email at jillian@dabbleofdevops.com and I will correct it.

1.2 Acknowledgements

There are truly too many people to thank. I've been very lucky in my career to have wonderful bosses, mentors, and coworkers along with fascinating projects. The scientific computing community at large is a open and welcoming environment, and overall they are a very fun bunch of people.

I would specifically like to thank the [Sphinx] project, which this book is written in. There are many other scientific libraries that use Sphinx as their documentation engine, so I feel that I am in good company. Images are either from their respective projects with the appropriate links, or created with LucidCharts.

1.3 About the Author

I'm Jillian Rowe, and I've been working in Bioinformatics for longer than I am willing to admit to anymore. I got my Masters in Bioinformatics at what is now the Tandon School of Engineering at NYU, spent about another 10 years in academia, and have been a full time independent consultant working for various biotech startups since.

I've worked on large and small scale Bioinformatics projects and have watched the field grow in complexity and maturity. My main interests these days are optimizing analyses for HPC systems and Apache Airflow, along with designing infrastructure around real time visualization of large scale genomics datasets with Dash or RShiny.

When I'm not working I like to be outdoors, baking, drawing, hanging out with my kids, or some combination of those. You can learn more about me, or about hiring me to work with your company or startup, on my website or by emailing me at jillian@dabbleofdevops.com.

CHAPTER

TWO

WHY?

Before we get into any more detail on AWS infrastructure solutions, let's talk a bit about the underlying reasons should we care about any of these.

Briefly, the entire why here is because we care about good science. More and more in the present day good science also relies on good data management and computational infrastructure, and so here we are. At the end of the day what we all want is a successful Bioinformatics project, which we will refer to as a Virtual Lab. A virtual lab could be:

- A new algorithm such as [BWA].
- A new analysis pipeline such as those developed by [Samtools] or [GATK].
- A new analysis software package such as [Seurat] or [CellProfiler].
- A new data visualization application such as [Scanpy].
- A new analysis pipeline or integration of several pipelines into a single press button solution that can be run by a workflow manager such as [Snakemake] or [ApacheAirflow].
- Research insights from a new dataset that will use one or more of the above. Often the findings will be discovered using one or more analysis libraries, and the results will be presented using one or more data visualization applications.

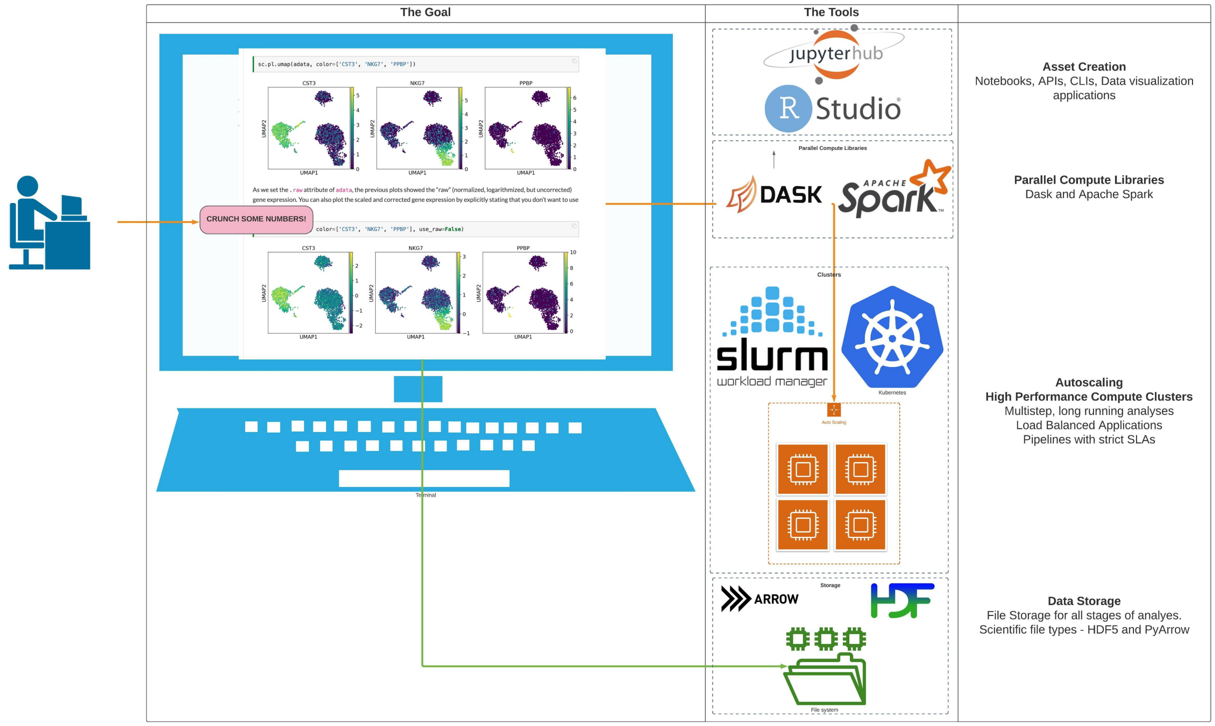
A successful Bioinformatics project can be consumed by academic research labs, industry R&D hubs, or biotech startups.

2.1 Benefits

Now that we know what we are going for, we can work backwards to think of what are the pieces that need to be in place in order to get at our goals listed above. One piece of this puzzle is to have a well thought out and standardized computing infrastructure. There are, of course there are other pieces, such as well designed experiments, project management, but we're only talking about compute infrastructure in this book.

- Create a standardized compute environment that is accessible by multiple people, along with new team members.
- Reduce lack of knowledge with churn. This is especially important in academic research where postdocs come and go.
- Encourage best practices for data management.
- Create Virtual Labs.

2.2 Virtual Labs


The Goal
The Tools
sc.pl.umap(adata, color=['CST3', 'NKG7', 'PPBP'])
CST3
NKG7
PPBP
UMAP1
UMAP2
As we set the .raw attribute of adata, the previous plots showed the "raw" (normalized, logarithmized, but uncorrected) gene expression. You can also plot the scaled and corrected gene expression by explicitly stating that you don't want to use
CRUNCH SOME NUMBERS!
color=['CST3', 'NKG7', 'PPBP'], use_raw=False)
Terminal
jupyterhub
R Studio
Parallel Compute Libraries
DASK
APACHE Spark
Clusters
slurm workload manager
Kubernetes
Auto Scaling
Storage
ARROW
HDF
File system
Asset Creation
Notebooks, APIs, CLIs, Data visualization applications
Parallel Compute Libraries
Dask and Apache Spark
Autoscaling
High Performance Compute Clusters
Multistep, long running analyses
Load Balanced Applications
Pipelines with strict SLAs
Data Storage
File Storage for all stages of analyes.
Scientific file types - HDF5 and PyArrow

Whenever I think about Bioinformatics Infrastructure, I always think of it as serving a purpose. That purpose is to create a Virtual Lab. A Virtual Lab is the result of a successful Bioinformatics project and almost exclusively contains one or more of the following (broken up into several categories). We will go into more detail in each of these categories in the next section. In the next chapter we will demonstrate what each of these means in a real world context.

Data Storage and Access

- Backed up, version controlled, annotated datasets
- Standardized machine readable file formats
- SOPs for data analysis

Let's take a step back and ask why we should care about any of these. Mostly, we are future proofing our lives and making sure that when something does go wrong, because something always goes wrong, that we can recover from our mistakes. Let's say I am a data scientist who is not fully caffeinated and I do an incorrect join on some datasets, and I accidently overwrite an existing dataset. If I have versioning enabled this is not a problem because I can simply revert to a previous state when I realize that my dataset doesn't make any sense. Better yet, let's say that I have a standardized machine readable format for my data. That data format will often have one or more baked in sanity checks, and when I try to write my dataset I may get an alert that all is not well in data wrangling land.

Also, if I have an SOP for data analysis and the creation of new datasets, the steps I took to create this erroneous dataset will be documented and understood to prevent future mistakes, and to verify that this is not a correct dataset and should be tossed.

Computing

- Standardized development environments, software stacks
- APIs for annotated datasets

I'm sure most people have run into problems with software stacks and conflicting versions at one point or another in their analysis career. I've even heard of months of time or work being lost because a team switched from one version of a software to another mid analysis, which changed their findings enough to result in an artificial batch effect. At best, dealing with software stacks is an annoyance, but at worst it can result in poor or incorrect research.

Research and Community Outreach

- APIs and frameworks for data analysis
- Data Visualization applications
- Use of standardized data formats, SOPs, and analysis libraries to allow for the sharing of data and analysis with data portals and collaborators.

Now, this is not to say that each projects needs each of these. Many successful projects excel in a single area, such as [BioConda] for software management, or [AnnData] for These are just the markers that have stood out to me the most when working on projects. This chapter is mostly about the science, and we will get into software management more in a later chapter.

When exploring an analysis it is nearly always preferable to have some existing framework for your data analysis, at least as a starting point. Sometimes these are generalized tools for manipulating data frames [Tidyverse] for R or [Pandas], [Numpy], [ScikitLearn] or [Jupyterhub] for Python. Sometimes these are entire programmatic analysis libraries such as [Seurat] for analyzing single cell data or [Plink] for whole genome analysis.

2.3 Data Science Community

I also want to make an additional quick point here. There has been a big push from the data science community to have sets of standardized tools that play nicely together and are a part of the same ecosystem. The [PyData] and [NumFocus] organizations have been instrumental in these efforts by supplying support and even funding. This also means, that generally speaking, if an analysis tool falls into the PyData ecosystem that it has a robust toolkit for data representation, storing in one or more machine readable formats, querying, mathematical and statistical operations, visualization, and more. None of this would by possible without the amazing data science community, which I am very grateful to have been a part of.

CHAPTER

THREE

SUCCESSFUL BIOINFORMATICS PROJECTS IN THE WILD

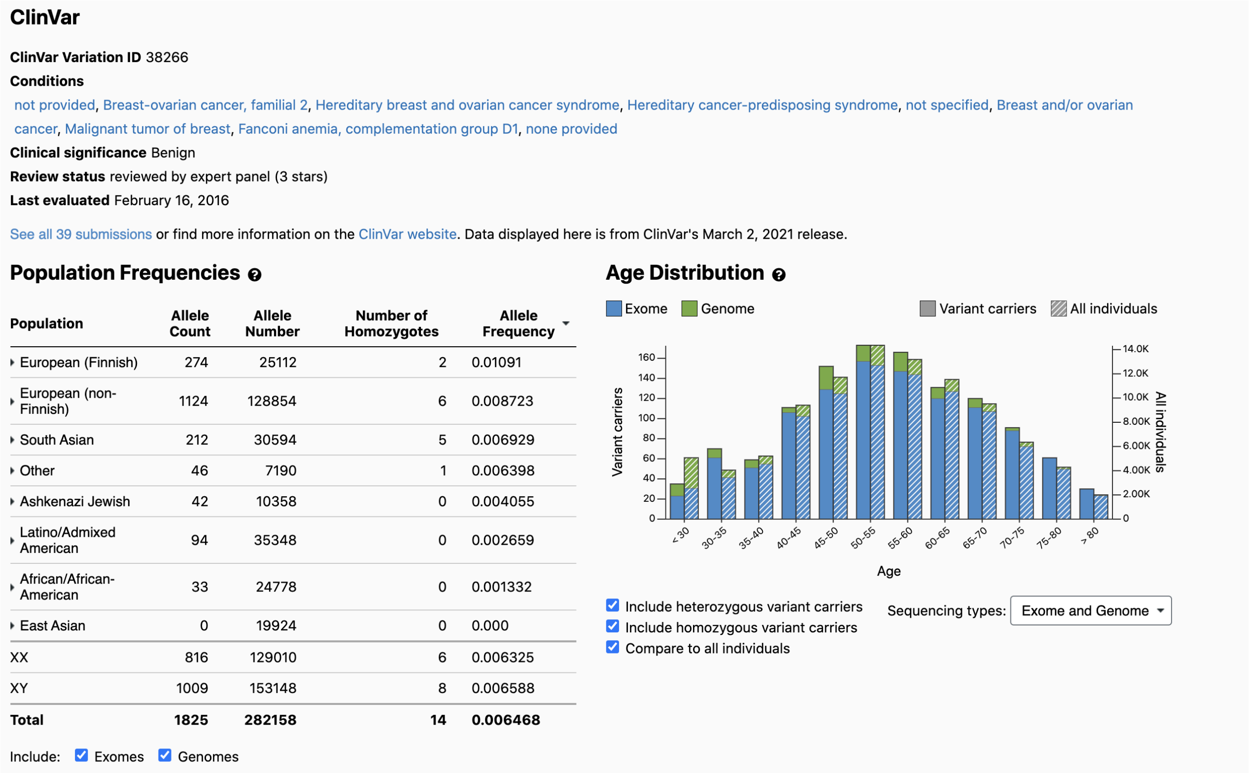
We've talked a lot about the conceptual idea of a Virtual Lab. Now let's talk about real projects. As a disclaimer, I am not directly involved with any of these projects, with the exception of one small shameless self promotion. These are simply projects I have used some part of, and I think are fairly widely known and applicable to anyone working in Bioinformatics.

I'm making an exception to my no code rule here for the purposes of illustration, but be aware that this is Bioinformatics, and that all libraries move quickly and introduce function changes regularly.

All credit for the code, examples, and visualization go the the authors of the libraries.

3.1 The gnomAD Browser from the Broad

> *The Genome Aggregation Database (gnomAD) is a resource developed by an international coalition of investigators, with the goal of aggregating and harmonizing both exome and genome sequencing data from a wide variety of large-scale sequencing projects, and making summary data available for the wider scientific community.*
>
> *The gnomAD Browser from the Broad https://gnomad.broadinstitute.org/*



ClinVar

ClinVar Variation ID 38266

Conditions

not provided, Breast-ovarian cancer, familial 2, Hereditary breast and ovarian cancer syndrome, Hereditary cancer-predisposing syndrome, not specified, Breast and/or ovarian cancer, Malignant tumor of breast, Fanconi anemia, complementation group D1, none provided

Clinical significance Benign

Review status reviewed by expert panel (3 stars)

Last evaluated February 16, 2016

See all 39 submissions or find more information on the ClinVar website. Data displayed here is from ClinVar's March 2, 2021 release.

Population Frequencies ?

Population	Allele Count	Allele Number	Number of Homozygotes	Allele Frequency ▾
▸ European (Finnish)	274	25112	2	0.01091
▸ European (non-Finnish)	1124	128854	6	0.008723
▸ South Asian	212	30594	5	0.006929
▸ Other	46	7190	1	0.006398
▸ Ashkenazi Jewish	42	10358	0	0.004055
▸ Latino/Admixed American	94	35348	0	0.002659
▸ African/African-American	33	24778	0	0.001332
▸ East Asian	0	19924	0	0.000
XX	816	129010	6	0.006325
XY	1009	153148	8	0.006588
Total	**1825**	**282158**	**14**	**0.006468**

Include: ☑ Exomes ☑ Genomes

Age Distribution ?

☑ Include heterozygous variant carriers
☑ Include homozygous variant carriers
☑ Compare to all individuals

Sequencing types: Exome and Genome ▾

Now, let's break down the different components of a successful Virtual Lab we see here.

Data Storage and Access

If you to the gnomAD downloads page you will see a variety of datasets available in a variety of formats. Each dataset is several stored on Google Cloud with [GCS], AWS with [S3], and Azure as an Azure Data Storage. Both [S3] and [GCS] are object stores, which have build in versioning and programmatic access.

Additionally, each dataset is stored in 2 different machine readable formats, [VCF] and [Hail]. Each of these is a machine readable format, with specs and sanity checks and all the goodies we like to see in our data formats.

Data Analysis

The datasets can be read in directly from their cloud storage data formats and analyzed using [Hail]. For example, here is a code snippet I completely grabbed from the very informative blog post Intro to Hail written by Kumar Veerapen.

```
# read in dataset from a machine readable format on gcs
gnomad='gs://gnomad-public/release/3.0/ht/genomes/gnomad.genomes.r3.0.sites.ht'
gnomad_ht = hl.read_table(gnomad)
# analyze using an analysis API exposed by the Hail software
mt = mt.annotate_rows(gnomad=gnomad_ht[mt.locus, mt.alleles])
```

Reading and writing data to/from the Hail format will provide necessary sanity checks and create a structure to our datasets. This structure is then consumed by an API provided by the [Hail] software that provides functions for statistical analysis, annotations, plotting, and more. Many publically available datasets then go on to power the [gnomAD] browser, which is a resource that is used by many labs and research institutes.

Data Visualization

Hail includes a set of functions for creating data visualizations on top of a Python visualization library [Bokeh]. Here is another example from the excellent Intro to Hail blog post.

```
p = hl.plot.scatter(x=mt.sample_qc.dp_stats.mean,
                    y=mt.sample_qc.call_rate,
                    xlabel='Mean DP',
                    ylabel='Call Rate',
                    size=8)
show(p)
```

Which results in:

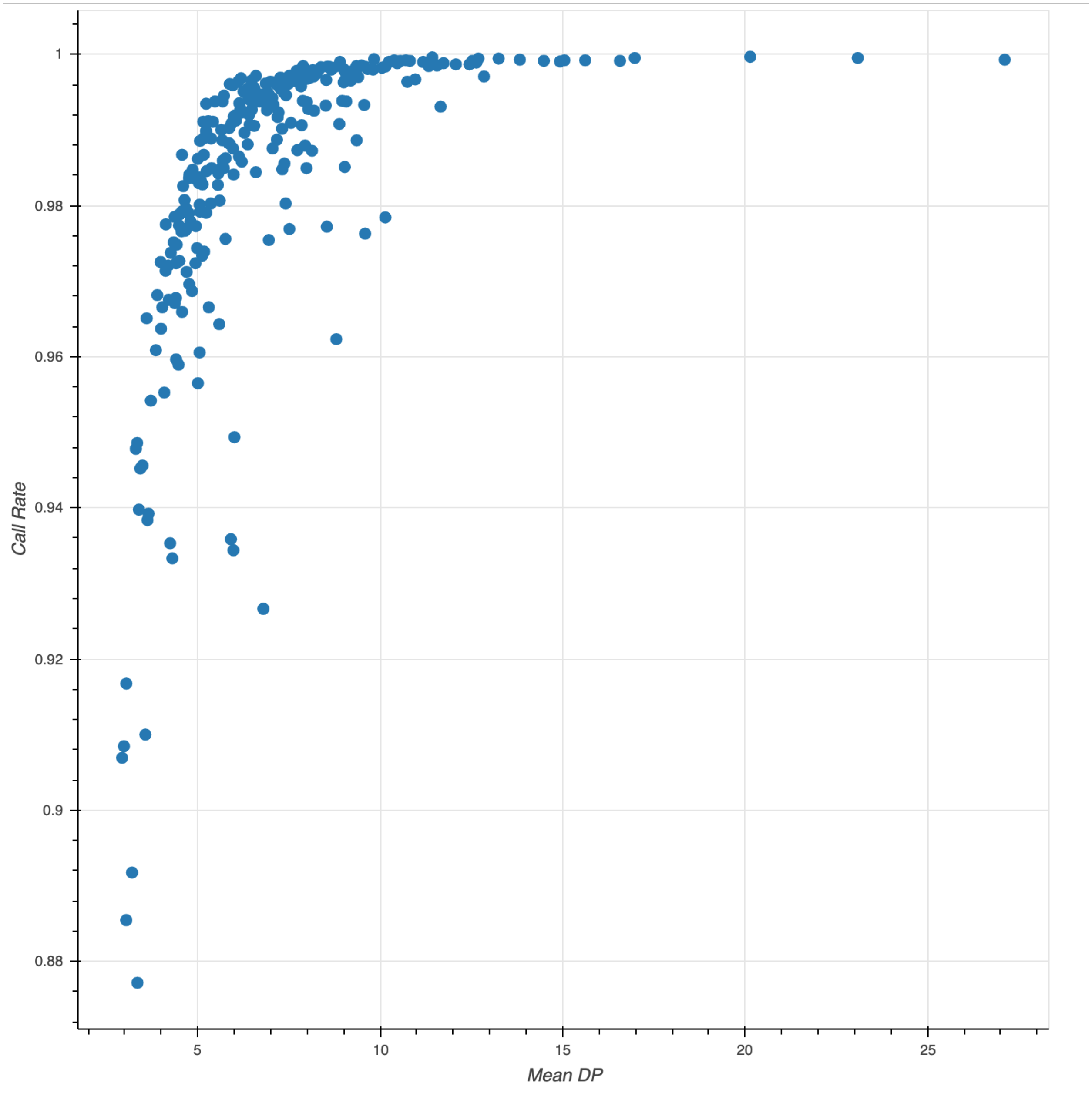
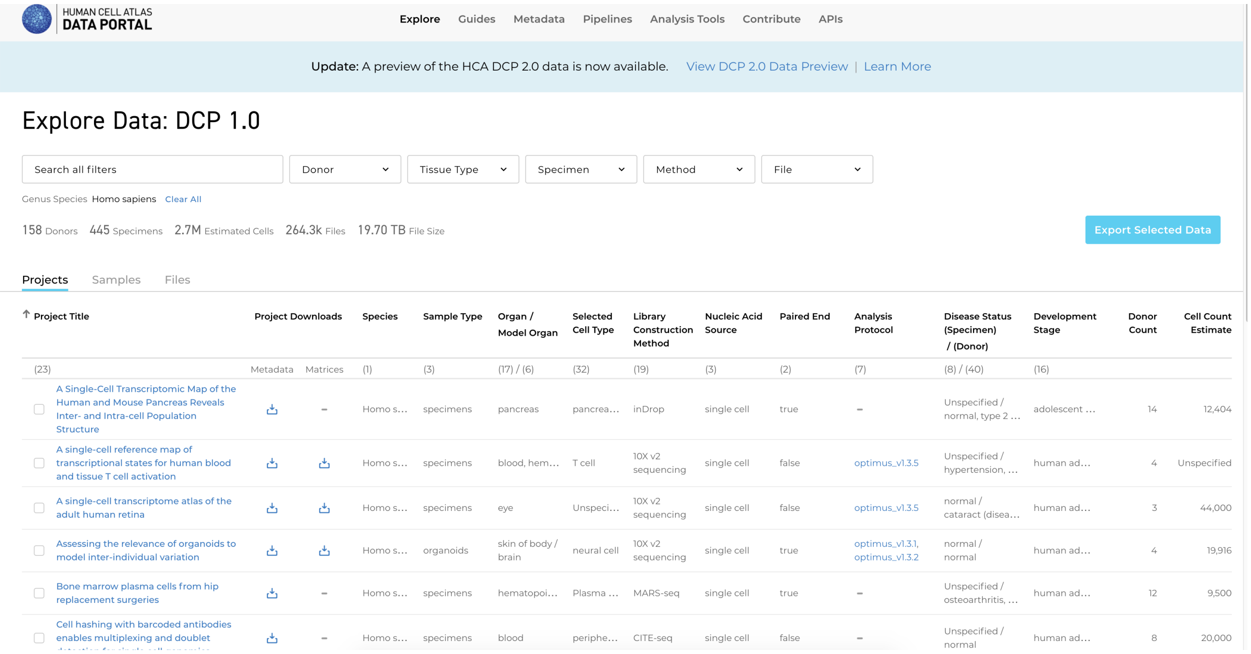


Image from https://blog.hail.is/introtohail/. All credit should go to the Hail project.

There is also the [gnomAD] browser, which is a data portal built from publically available datasets from the Human Genome Project ([HumanGenomeProject]) and stored with [Hail].

3.2 The Human Cell Atlas

> *About the Human Cell Atlas In London on 13 and 14 October, 2016, a collaborative community of world-leading scientists met and discussed how to build a Human Cell Atlas—a collection of maps that will describe and define the cellular basis of health and disease.* Human Cell Atlas



Update: A preview of the HCA DCP 2.0 data is now available. View DCP 2.0 Data Preview | Learn More

Explore Data: DCP 1.0

Search all filters | Donor | Tissue Type | Specimen | Method | File

Genus Species Homo sapiens Clear All

158 Donors 445 Specimens 2.7M Estimated Cells 264.3k Files 19.70 TB File Size

Export Selected Data

Projects Samples Files

	↑ Project Title	Project Downloads		Species	Sample Type	Organ / Model Organ	Selected Cell Type	Library Construction Method	Nucleic Acid Source	Paired End	Analysis Protocol	Disease Status (Specimen) / (Donor)	Development Stage	Donor Count	Cell Count Estimate
	(23)	Metadata	Matrices	(1)	(3)	(17) / (6)	(32)	(19)	(3)	(2)	(7)	(8) / (40)	(16)		
□	A Single-Cell Transcriptomic Map of the Human and Mouse Pancreas Reveals Inter- and Intra-cell Population Structure	⤓	–	Homo s…	specimens	pancreas	pancrea…	inDrop	single cell	true	–	Unspecified / normal, type 2 …	adolescent …	14	12,404
□	A single-cell reference map of transcriptional states for human blood and tissue T cell activation	⤓	⤓	Homo s…	specimens	blood, hem…	T cell	10X v2 sequencing	single cell	false	optimus_v1.3.5	Unspecified / hypertension, …	human ad…	4	Unspecified
□	A single-cell transcriptome atlas of the adult human retina	⤓	⤓	Homo s…	specimens	eye	Unspeci…	10X v2 sequencing	single cell	false	optimus_v1.3.5	normal / cataract (disea…	human ad…	3	44,000
□	Assessing the relevance of organoids to model inter-individual variation	⤓	⤓	Homo s…	organoids	skin of body / brain	neural cell	10X v2 sequencing	single cell	true	optimus_v1.3.1, optimus_v1.3.2	normal / normal	human ad…	4	19,916
□	Bone marrow plasma cells from hip replacement surgeries	⤓	–	Homo s…	specimens	hematopoi…	Plasma …	MARS-seq	single cell	true	–	Unspecified / osteoarthritis, …	human ad…	12	9,500
□	Cell hashing with barcoded antibodies enables multiplexing and doublet	⤓	–	Homo s…	specimens	blood	periphe…	CITE-seq	single cell	false	–	Unspecified / normal	human ad…	8	20,000

The Human Cell Atlas ([HumanCellAtlas]) is a truly cool project. It is less of a project and more of an ecosystem for best practices around Single Cell research. It hits what I consider to be all the spots of a truly successful data science project, from data access with machine readable formats, all the way up to interactive data visualization with a specialized cell browser.

Data Storage and Access

Much like we saw when dissecting the [gnomAD] browser and [Hail] software, we see a similar pattern here. Datasets are defined by a very particular data structure, which is managed by a software called [AnnData]. The data is stored in a machine readable format, [HDF5].

Here is an example from the Scanpy Tutorial.

```
adata = sc.read_10x_mtx(
'data/filtered_gene_bc_matrices/hg19/',  # the directory with the `.mtx` file
var_names='gene_symbols',                # use gene symbols for the variable names (variables-axis index)
cache=True)                              # write a cache file for faster subsequent reading
```

Analysis APIs

The single cell data is stored in a data format described by [AnnData] that is consumed by the analysis software [Scanpy]. Here is an example from Scanpy Tutorial showing how to get the genes with the highest expression in the dataset.

```
sc.pl.highest_expr_genes(adata, n_top=20, )
```

Here is another example showing how to use the [Scanpy] library to get a PCA plot from dataset.

```
sc.tl.pca(adata, svd_solver='arpack')
```

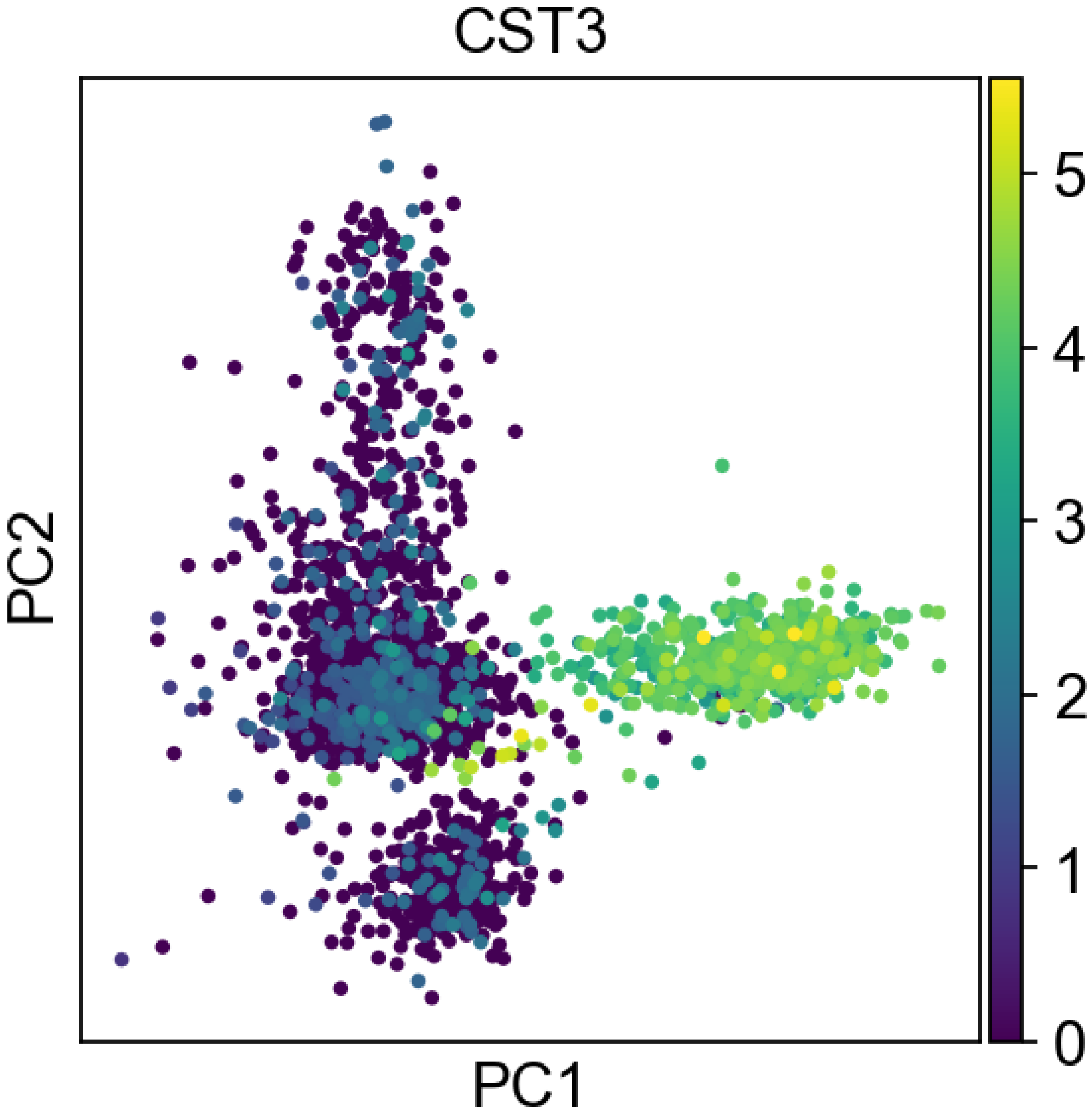


Image from the Scanpy Tutorial https://scanpy-tutorials.readthedocs.io/en/latest/pbmc3k.html`_

This brings us to an important point I briefly discussed earlier in the Why chapter. Although the data is formatted using a custom library, [AnnData], underneath the hood it is using libraries that are all a part of the [PyData] ecosystem such as [Numpy] for representing numeric and matrix data along with [Pandas] for more descriptive data. Since the data is written to [HDF5] it can be stored on a local filesystem or cloud object store such as [S3]. Much of the internal plotting demonstrated by the tutorial is done using wrappers around the data API and displayed using [Matplotlib].

Visualization

Once all the pieces are in place for data storage and data analysis some truly cool solutions start popping up. For the last few years I have become increasingly interested in real time data visualization of large datasets using scientific libraries and web servers. The CellxGene is an excellent example of that. It takes datasets managed with [AnnData] and analyzed with [Scanpy], and allows for real time data visualization and interactive analysis of data. You can read, write, cluster, and visualize your data all in a browser using [CellxGene].

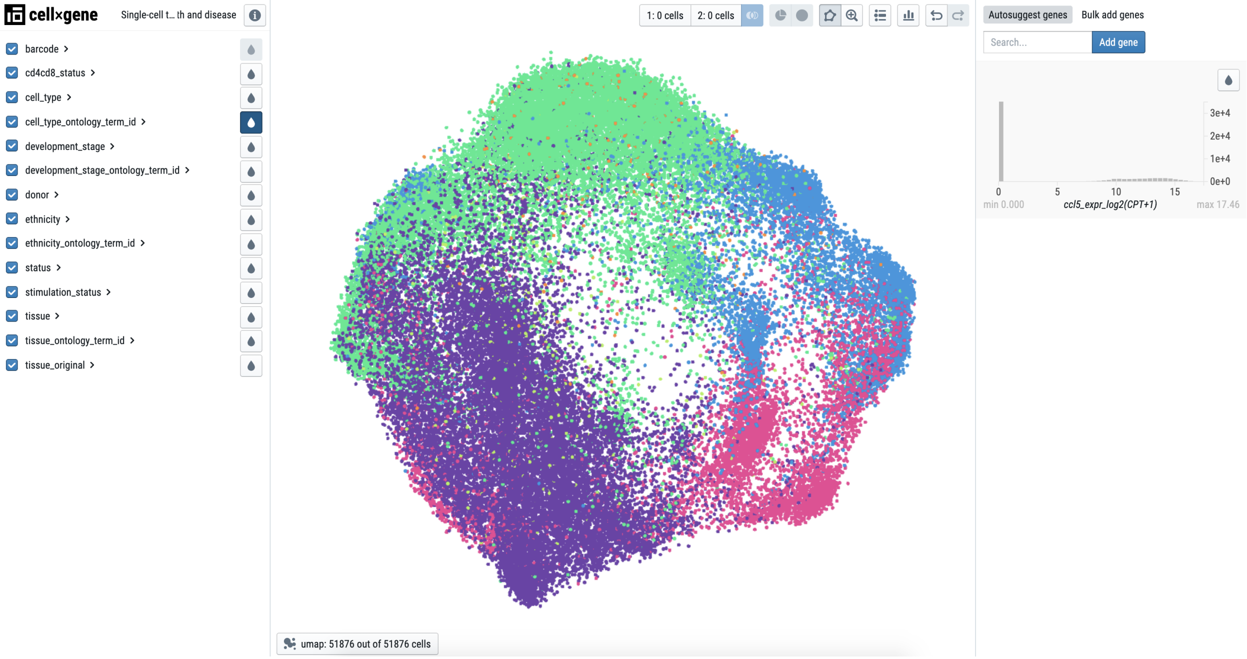


Image from the CellxGene Data Portal - https://cellxgene.cziscience.com/e/01209dce-3575-4bed-b1df-129f57fbc031.cxg

Community

The Human Cell Atlas is very much a project with a heavy emphasis on community. It hosts data sets and analysis portals for hundreds of data sets, and allows for anyone to download any of their datasets and tools.

3.3 CellProfiler

[CellProfiler] is a Cell Image Analysis software that is especially interesting because it has really stood the test of time. It was started before many of the [PyData] libraries were available or standardized. They also have an excellent outreach program and put a heavy emphasis on designing their software for use by Biologists.

The team has done an incredible job lately of updating their libraries to conform with the latest from the [PyData] ecosystem, including using [Numpy] for it's numeric data and integrating with the [ScikitLearn] library for much of it's analysis and image processing. This included some very cool work to refactor their library to allow for the codebase to be used within the CellProfiler GUI, or by using [Jupyterhub] as a GUI to run analyses programatically. Previously, all work had to be done using their desktop application, and desktop applications do not always play nicely with large datasets or High Performance Compute Environments. All in all, this was a very cool move, and one I was very happy to see!

Data Storage and Access

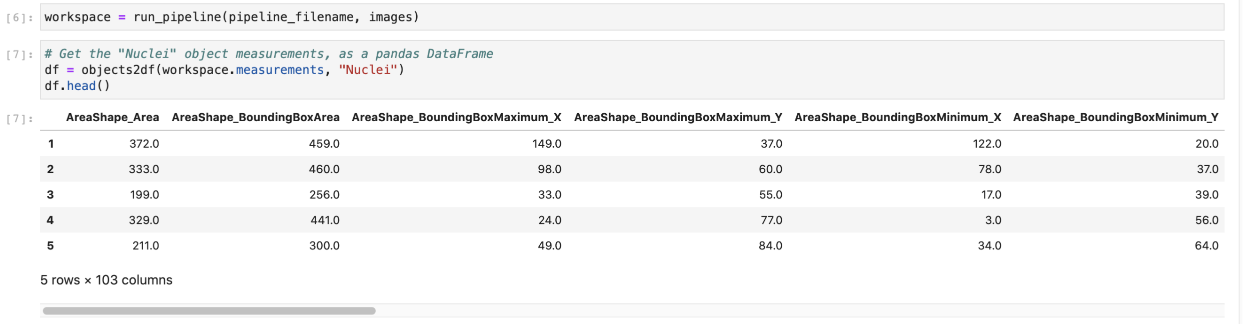
The data fed into CellProfiler needs no particular format as it is cell data that is captured by a microscope and stored as an image. Analysis pipelines, however, can become quite complex and are stored as *.cppipe* files in [HDF5].

Data Analysis

I'm really going to break my no code rule, but it's for a worthy cause. This code was adapted from the Cellprofiler Notebooks in the github repo and updated for the most recent version. Data is the same as that shown in the repo. To see the functions, including setting up the analysis functions, please see the demo notebook.

```
import cellprofiler
import cellprofiler_core
import cellprofiler_core.pipeline
from cellprofiler_core.preferences import set_headless
import cellprofiler_core.image
import cellprofiler_core.measurement
import cellprofiler.modules.maskimage
import cellprofiler_core.pipeline
import cellprofiler_core.workspace
###
# for the sake of brevity not all the analysis code is here
# ...
data_dir = os.path.join(os.getcwd(), "data")
images = {
    "OrigBlue": skimage.io.imread(os.path.join(data_dir, "images/01_POS002_D.TIF")),
    "OrigGreen": skimage.io.imread(os.path.join(data_dir, "images/01_POS002_F.TIF")),
    "OrigRed": skimage.io.imread(os.path.join(data_dir, "images/01_POS002_R.TIF"))
}
pipeline_filename = os.path.join(data_dir , "ExampleFly.cppipe")
# Run the analysis!
workspace = run_pipeline(pipeline_filename, images)
# Get the "Nuclei" object measurements, as a pandas DataFrame
df = objects2df(workspace.measurements, "Nuclei")
df.head()
```

Now, what we have here is the CellProfiler library, that gives us programmatic access to a set of analyses that are stored in [HDF5] files. From there, we can run the analyses, and use the library to various results as a [Pandas] dataframe. As a quick side note dataframes's can be read to and from databases with simply the [Pandas] library using the *to_sql* package, which makes it an excellent choice for those that want to persist their data to a database. Because Cellprofiler plays nicely with the PyData ecosystem we get oodles of additional functionality, including all the statistics we can generate directly from a dataframe, a querying language, etc.



```
[6]: workspace = run_pipeline(pipeline_filename, images)
```

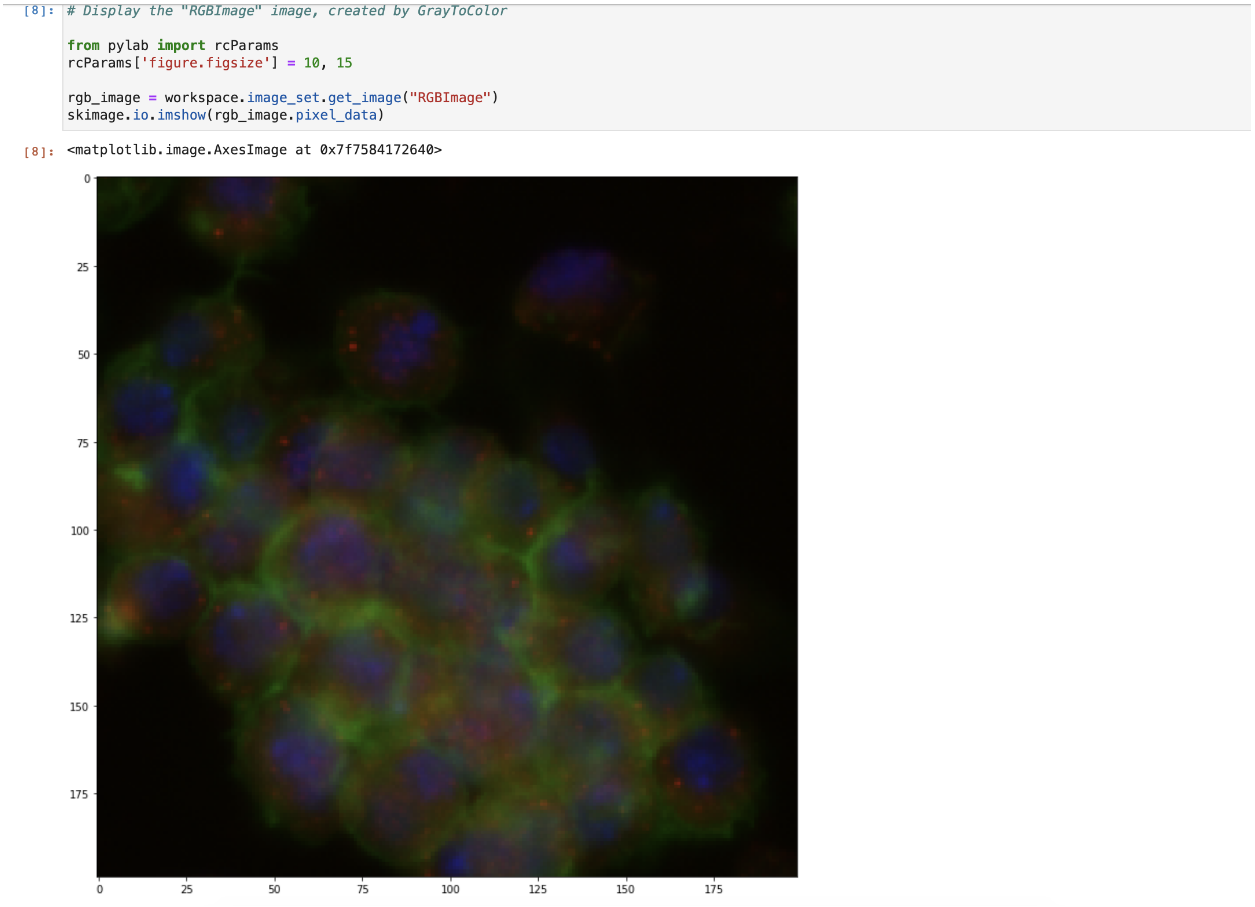
```
[7]: # Get the "Nuclei" object measurements, as a pandas DataFrame
     df = objects2df(workspace.measurements, "Nuclei")
     df.head()
```

[7]:

	AreaShape_Area	AreaShape_BoundingBoxArea	AreaShape_BoundingBoxMaximum_X	AreaShape_BoundingBoxMaximum_Y	AreaShape_BoundingBoxMinimum_X	AreaShape_BoundingBoxMinimum_Y
1	372.0	459.0	149.0	37.0	122.0	20.0
2	333.0	460.0	98.0	60.0	78.0	37.0
3	199.0	256.0	33.0	55.0	17.0	39.0
4	329.0	441.0	24.0	77.0	3.0	56.0
5	211.0	300.0	49.0	84.0	34.0	64.0

5 rows × 103 columns

Additionally, because CellProfiler is using [ScikitLearn], we get all the functionality from that library, including the Sklearn Image library. We can view any image, using different channels and color formatting with the library.



```
[8]: # Display the "RGBImage" image, created by GrayToColor

     from pylab import rcParams
     rcParams['figure.figsize'] = 10, 15

     rgb_image = workspace.image_set.get_image("RGBImage")
     skimage.io.imshow(rgb_image.pixel_data)
```

```
[8]: <matplotlib.image.AxesImage at 0x7f7584172640>
```

Viewing [Pandas] data frames, [Numpy] data structures, and images are all very well supported within [Jupyterhub].

3.4 Wrap Up

This wraps up our examples of Bioinformatics projects out in the wild. There are, of course, many such projects that could be cited here. For the sake of not making this book an encyclopedia I have kept it to 3 examples. I hope you can see how important having each of your infrastructure pieces in place is to have a wildly successful project.

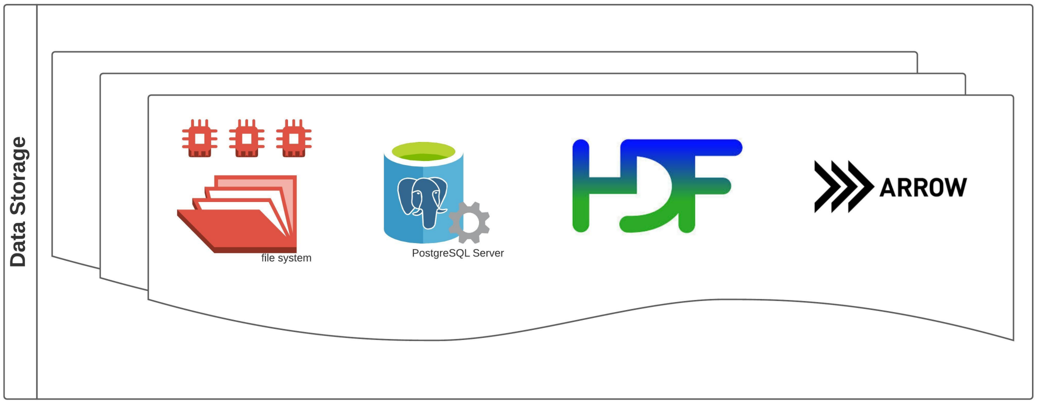
CHAPTER

FOUR

COMMON INFRASTRUCTURE COMPONENTS

In this chapter we'll discuss the most common infrastructure components you'll encounter. Each of these components is used across many AWS services and your understanding of these will be vital in understanding your own infrastructure.

4.1 Storage



No matter what kind of analyses you're running you'll need access to storage to persist data, install software, and generally share the types of resources that one expects to see in bioinformatics such as alignment files, QC reports, along with an abundance of analysis scripts and configurations.

4.1.1 S3

AWS [S3] stands for Amazon Simple Storage Service. It is a pay as you use service for creating what called buckets, and share many similarities to NFS file systems. S3 is also a popular choice among web developers because it can host static html sites, such as documentation sites like those seen at read the docs, along with being a popular choice for data scientists running cloud native pipelines.

S3 is also a popular choice for many other AWS Services. For example, if you run a managed PostgreSQL database on AWS, you can set it to automatically backup to S3.

AWS S3 is available anywhere you have access to the AWS CLI and your authentication values, making it an excellent choice for data scientists who do not want to have to be concerned over mounting file systems. Many groups will also publically host data with S3.

AWS S3 has also been gaining popularity with the Scientific Python community. Many scientific file formats such as [Zarr], [HDF5] along with interfaces on top of these file formats such as [Xarray] allow for files to be stored on S3 along with local file storage.

Many of these file types are compatible with [Dask], a parallel computational library, allowing for extra shininess with chunking, and compression. Dask also supports lazy reading, writing, and computation, meaning you don't have to read the entire dataset into memory (vi very-large-vcf file anyone?).

The solutions I mentioned are Python specific, but R users can benefit from any of the solutions in the [ApacheSpark] ecosystem such as [ApacheParquet] files.

Each bucket can be configured independently of the others. Configuration values include

- **Privacy settings as ACLs**
  - Public, Private, Public to certain IP addresses or AWS users
  - Public/Private Read/Write
- Lifecycle Policies

The main drawback to using S3 is that although it mostly feels like a filesystem it is not entirely a filesystem. You can't use symbolic links, so trying to create a software stack with conda and uploading the whole thing to S3 is unlikely to work. Unless you're using a solution like S3Fuse, or a python library to access your data files remotely, you'll need to be concerned with pushing and pulling data from the bucket. This can feel unnatural to many scientists who are used to working on a cluster with a local or networked file system, and you're likely to run into other small gotchas when trying to hammer S3 into working as a file system does.

Popular Projects using S3

- [S3FS] Dask S3FS - Access your S3 files in python
- [S3Fuse] S3Fuse - Mount an S3 bucket on Mac or Linux with Fuse.
- [DistributedCellProfiler] Distributed CellProfiler - A project to access the very popular Cellprofiler package in a distributed manner on AWS. It uses S3 as it's backend storage for configuration and S3Fuse to make those files accessible to compute nodes.
- [AnnData] AnnData - In their own words, a project that provides a scalable way of keeping track of data and learned annotations, using an Xarray backend that supports HDF5 and Zarr files.

4.1.2 Managing S3 Costs with Lifecycle Policies

S3 has several options for how often you need to access the data, known as storage classes. At the time of writing the storage classes from highest to lowest are:

- Standard
- Intelligent-Tiering
- Standard-IA (Standard Infrequent Access)
- One Zone IA
- Glacier
- Glacier Deep Archive

The lower the storage class the lower the cost.

Like all services on AWS these can be managed manually or through specific configurations, in this case lifecycle policies. For example, let's say we want an automatic archival system. While the data is being actively analyzed you'll need to be able to access it as quickly as possible, but after the data has been analyzed it's safe to place it in infrequent access or archival storage.

You could say that after a file (known as an object) has been in S3 for over 30 days to transfer it from standard access to glacier. In this way AWS automatically manages your data for you, leaving you with more money and more free time.

Resources

- AWS S3 Storage Classes
- Configure S3 Bucket Lifecycle Policies in Terraform

4.1.3 Networked File Storage - EFS

I often like to joke that no matter how fancy your devops solutions may be, that really we're just all running around configuring a bunch of servers and networked file storage with bash.

Then I tell people that their infrastructures are run with bash, find, and grep, and they think I am significantly less funny.

Most Bioinformaticians and Data Scientists are familiar with NFS file systems. They are likely present as */data* or */scratch* if you've worked on a multiuser HPC system before.

The AWS solution to NFS is [EFS] Elastic File Storage. Remember what I said about looking for solutions that have elastic in their title? This is a prime example of an elastic service. When creating an EFS service you do not need to declare how much storage you need upfront. AWS will grow with the amount of data stored on the file system. You can use any of the scientific file types I mentioned earlier, such as [Zarr], [HDF5], or [NetCDF].

Beyond that EFS is, well, a file system. You can add folders, files, symlinks, install Conda packages, rsync, and apply user and group permissions just as you would on any linux system.

EFS can be mounted to any EC2 instance in the same VPC. With some configuration you can mount it to EC2 instances in other VPCs. This means it can be mounted to any of the compute solutions I'll discuss later in this book, such as Kubernetes or HPC clusters with SLURM or AWS Batch.

4.1.4 Managing Storage with Lifecycle Policies

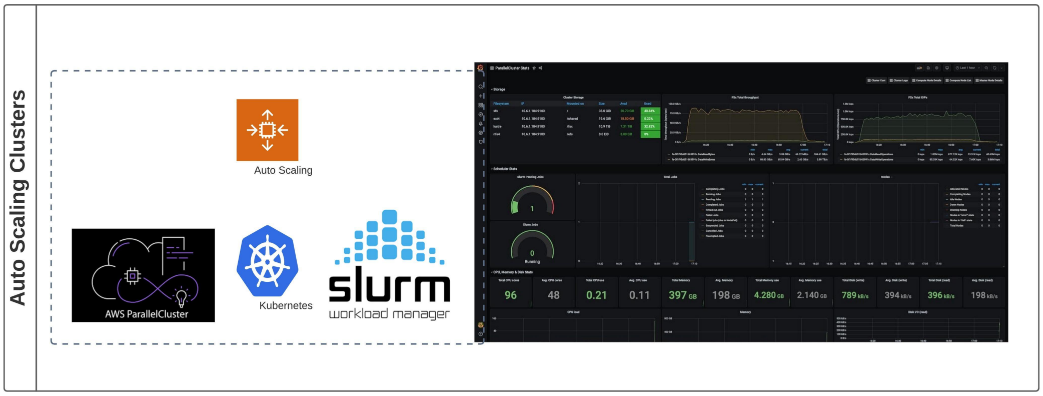
EFS comes in two flavors, very expensive, or Standard Access, and just expensive, Infrequent Access. In order to cut down on costs you can set a lifecycle policy to move data from Standard Access to Infrequent Access after a period of 7, 14, 30, 60 or 90 days. These policies are managed for automatically by AWS and the policies you set.

Another way to manage costs is to store only the data you are analyzing *right now* on EFS, then archive it to S3 Glacier. At the time of writing prices start at $1/terrabyte/month!

Resources

- AWS EFS Lifecycle Policies
- Terraform Configuration for EFS with Lifecycle Policy
- AWS S3 Glacier

4.2 Compute Servers



We all know that Bioinformaticians spend most of their time clobbering linux servers and *bending them to our collective wills*. We'll get into your options here.

4.2.1 Individual Servers - EC2

EC2's are the servers, Linux or Windows, of the AWS world. They are managed the same way you would manage a server not on AWS. EC2 instances are built on a range of OSes and you can choose from a large array of memory, cpu, memory optimized or not.

Getting an EC2 instance is very straight forward. You can log into the AWS console, select EC2 as a service. The wizard will prompt you to select an AMI type, which includes an OS and a version, an instance type, which includes the amount of memory and CPU. You can also choose how much local storage, EBS to use, which is a type of storage we didn't cover, but comes as the default on EC2.

Once your server is up and running you can do anything you would normally do with any server. You can install software, webservers, and run as many pipelines as you have storage and compute power for.

Resources

- Getting Started with EC2 Tutorial from AWS

4.2.2 Managed Elastic Compute Clusters

This is where the magic happens, and what we will be discussing for the rest of this book. If there is a cluster you have thought of it is probably available on AWS.

I am notably biased and think the best solutions for computation are auto scaling clusters. These get you piece of mind that you will have compute power when you need it, and don't require human intervention to bring up or down.

For the rest of this book we will be discussing 3 main types of clusters. First, Kubernetes, which is a cluster for deploying mainly web applications, but has also been embraced by the scientific community and has well supported solutions for [JupyterhubOnKubernetes], [ApacheSpark], and [Dask]. [Bitnami] has also decided to offer many of their applications as Helm charts, which is the Kubernetes package manager. Second, is a traditional HPC cluster, such as a SLURM or SGE cluster. The third type of cluster is an AWS Batch cluster, which is the AWS solution for HPC. Each cluster type has it's own pros, cons, and analysis areas where it excels.

Once you get to the point where you need to scale your application or cluster you will need to carefully consider your infrastructure and chosen AWS services. Backing each of these services, and other services we will discuss later, is the idea of an [AutoScaling] which brings up or spins down [EC2] instances based on a set of predefined rules.

Resources

- [Bitnami] - You can deploy *all the things* with Bitnami.
- [JupyterhubOnKubernetes] - Deploy an autoscaling Jupyterhub Hub on Kubernetes.
- [AWSParallelCluster] - Deploy HPC on AWS.

4.2.3 Managing Compute Costs

This one is slightly more tricky than just setting a lifecycle policy. There are several ways that you can deal with costs, and it just depends on what you are doing.

The simplest, but probably most difficult to implement, is to make sure you're only using what you need. In my opinion, this means using autoscaling clusters along with data lifecycle policies, instead of long running servers, because they will automatically spin down when a job ends or an instance is seen as idle. Each of these clusters uses EC2 instances on the backend, with an additional API or program layered over to take care of autoscaling and culling idle instances.

Along with that you buy reserved instances. Instead of paying for your EC2 instances on demand you pay upfront, but in return you save quite a bit, up to 70%, on the on demand costs.

Then there is configuring spot instances. Spot instances have a signifcant price discount, up to 90% off the on demand price, because you use whatever EC2 instances are left over. There is no guarantee that they will stick around, and can stop and get swapped in for another instance with little notice. This option is more difficult because it requires configuration that is not always straight forward. It will also only work out in very specific use cases, such as a stateless web application, or an analysis that can restart cleanly.

If you'd like to stick to single EC2 instances, you can also create Cloudwatch Alarms, which can stop your instance based on usage, such as CPU going below a certain threshold for a certain amount of time.

4.3 Cost Savings Tips

It's always a good idea to look for terms such as "auto-scaling", "elastic" or "on demand" when using AWS. This means that there is some smart mechanism baked in that only uses the resources when you need them, and this cuts down on costs.

The other major cost saving tip I am aware of is to buy reserved EC2 instances, which basically means you pay upfront for a month or a year, but at significant savings.

Resources

- Use Cloudwatch to create Alarms on a single instance
- Buy a reserved EC2 instance
- Configure Spot instances

CHAPTER

FIVE

COMMON DEPLOYMENT STRATEGIES

Remember how I said no code? Well, I *meant it*! Instead, what I will do here is to list some of my common haunts for various modules, recipes, and deployment strategies.

Mostly, I am exceedingly opportunistic. I'll google what I want to do, and see if there is a Terraform or Cloudformation recipe, or a tool built using either of those that I can hammer on. Which ever one most closely matches what I want and that I can most easily understand I'll use.

5.1 AWS Web Console

The first place any of us deploys much of anything is probably the AWS Web Console. It is extremely slick and under constant improvement. You can deploy and configure any service through the console.

There are several pros to using the console. First of all, it's there, and it's supported by AWS so it is highly unlikely that it has anything weird in there, parameters that reference previous versions of libraries, or any of the other gotchas we'll see in with the other choices. For the most part, the console is very intuitive. Many services have interactive form wizards that lead you through the process of deployment. For many scientists this will be enough.

Cons to using the web server include that it gets very difficult to keep track of multiple projects and deployments. Since its a web UI there is little in the way of automation. If you need to write Standard Operating Procedures (SOPs) for your deployments saying "I use the console" is unlikely to cut it, particularly if you are constantly deploying.

5.2 Infrastructure as Code

There are many infrastructure as code, configuration, as a service, etc tools and frameworks out there. Terraform and Cloudformation are the two I use most frequently.

5.3 Terraform

Terraform is an Infrastructure as Configuration that uses configuration files with syntactical sugar on top in order to deploy resources. These resources do not even necessarily have to be AWS, but can also include other cloud providers, such as Google Cloud or Azure, or even local services.

I personally find Terraform very easy to read, and sometimes when the AWS docs confuse me I will hop on over to the Terraform Docs to see if a certain configuration becomes any clearer to me. I'm fairly certain that says more about how long I've spent in front of a terminal than it does about any given documentation source, but you may find a similar case.

Terraform also has an extremely handy feature where it keeps track of your state, like a Makefile. Let's say you declare that you want an S3 bucket. You'd run the command to tell Terraform to do it's magic and create your S3 bucket. From there it keeps track of all sorts of important metadata about that bucket, such as it's name, region, any policies attached, whether or not it's versioned, etc. Because it keeps track of your state you can make changes to various configurations, such as changing an S3 bucket from private to public-read, all within Terraform. It knows that the bucket itself has already been created, so it will only change the attributes of the bucket.

Terraform has specific build recipes known as modules. You can create your own modules or use those that are already available. Since they are mostly just configuration files it you can usually configure these modules without too much trouble.

Terraform modules makes it very easy for you to share recipes and configure stacks. Let's say you want a dev and prod stack for a specific application you are building. This would be quite tedious to do through the UI, but with Terraform you could simply reference the module and input the correct variables.

It also makes it easier to separate out your infrastructure in terms of projects. Maybe you have a dev and prod RShiny server for single cell analysis. Then you also have a HPC SLURM cluster for large scale and exploratory analysis, along with an AWS Batch cluster for your production analyses. If you don't know what any of these are, don't worry, we'll talk about them later. Each of these recipes could be stored in a separate folder, referencing one or more modules or even custom configurations. These recipes can be stored, shared, and placed under version control and continuous integration and deployment.

As you can create stacks for specific projects so can you delete them! Deploying and deleting are each a single command.

Terraform allows for the execution of arbitrary commands with a bash executor. To be clear, they say that this is a hack and that you shouldn't use it, but I just can't help myself. I *love* this. The first thing I learn when I learn any new system is how to just get myself a shell so I can run what I want. For more practical purposes this means you can write tests, in [PyTest], Bats, for when I get sick of everything and just want bash and SSH, or another testing framework of your choice, to make sure that everything is as expected. I like to call these tests for dummies. You can also add in custom configurations, such as rsyncing files or running commands over SSH.

Finally, since Terraform is mostly configuration files, it is very easy to wrap it in a Cookiecutter project, or any other templating language for that matter, for more dynamic values and configurations. I would like to say I do anything truly fancy, but most often I simply indicate a boolean value for create EFS yes/no.

Resources

- AWS Terraform Modules
- AWS Terraform EKS Module - Managed Elastic Kubernetes on AWS
- CloudPosse DevOps Accelerator - A DevOps consulting company that creates excellent Terraform Modules.
- QHub - an open source project from Quansight that enables organizations to build and maintain cost-effective and scalable compute/data science platforms on-premise or on any cloud provider with minimal in-house experience.
- CookieCutter - Project templating library that I often couple with Terraform to create more dynamic modules.

5.4 Cloudformation

Cloudformation is the official AWS Infrastructure as code tool. It appears in many of their labs and tutorials, and there is a nice UI for building out the Cloudformation templates themselves.

One major perk of using Clouformation is that it's the closest thing to a project based view that the AWS console has. For example, you could create a Cloudformation template to deploy a Docker Swarm cluster, called *docker-swarm-cluster*. All the resources associated to that template would all come up for you under the Cloudformation UI under the *docker-swarm-cluster*. Like Terraform modules Cloudformation Stacks are created and destroyed as a unit, making them a very good fit for project specific infrastructure.

I don't think there is any clear winner between any of the Infrastructure as Code tools. I think it just depends on personal preference. Go take a look at the docs, and see which one you find to be most readable. Then use that one.

Resources

- Cloudformation Recipes on Github - Modules to deploy *all the things* on AWS.
- AWS Parallelcluster - A tool to deploy HPC clusters on AWS, built on top of Clouformation.
- Cloudformation Troposphere - Is a Python library to create AWS CloudFormation templates.

5.5 Considerations for Continuous Integration / Deployment (CI/CD)

Any solution on AWS is suited for CI/CD because AWS is built from the ground up to be automatically provisioned and deployed. You can automatically create and destroy Kubernetes Clusters, EC2 instances, Batch, or HPC clusters without knowing the intricacies of AWS once you have your requirements mapped.

5.6 Commercial Solutions

There are also several companies that offer consulting or full on SAAS or IAAS offerings for various deployments and analyses. Here is a certainly nonexhaustive list.

- Dabble of DevOps is my company that I run and I'm using this chance for shameless self promotion.
- Quansight does a kinds of cool things and has hands in just about all of the DataScience and PyData ecosystem. They also created Qhub which is an open source tool for deploying Kubernetes clusters.
- Coiled has a service offering to deploy Dask for everyone, everywhere.
- Bioinformatics CRO offers Bioinformatics Done Right, Now.

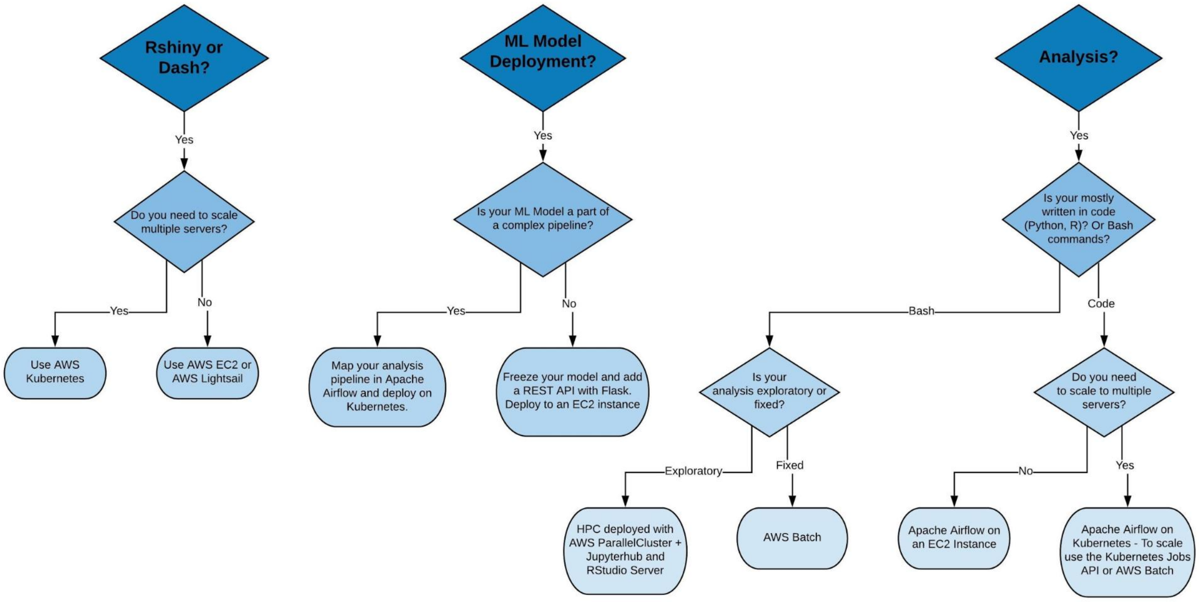
I have a fairly strict no jerks policy, so you can be assured everyone on the list is not a jerk.

CHAPTER SIX

OVERVIEW OF ANALYSIS TYPES

This chapter is meant to serve as a broad overview for different analysis types, or maybe categories. By types I do not mean their scientific types, such as RNASeq or single cell, but rather a way of categorizing analyses into their broadest computational needs.

Bioinformatics infrastructure on AWS includes the need for deploying data visualization applications such as [RShiny], [Dash], along workflow management frameworks such as [ApacheAirflow], [Snakemake], [Nextflow] or [Cromwell], capabilities for High-Performance Computing, along with the need for logging for general purposes, record keeping, and troubleshooting.


Rshiny or Dash?
Yes
Do you need to scale multiple servers?
Yes
No
Use AWS Kubernetes
Use AWS EC2 or AWS Lightsail
ML Model Deployment?
Yes
Is your ML Model a part of a complex pipeline?
Yes
No
Map your analysis pipeline in Apache Airflow and deploy on Kubernetes.
Freeze your model and add a REST API with Flask. Deploy to an EC2 instance
Analysis?
Yes
Is your mostly written in code (Python, R)? Or Bash commands?
Bash
Code
Is your analysis exploratory or fixed?
Exploratory
Fixed
HPC deployed with AWS ParallelCluster + Jupyterhub and RStudio Server
AWS Batch
Do you need to scale to multiple servers?
No
Yes
Apache Airflow on an EC2 Instance
Apache Airflow on Kubernetes - To scale use the Kubernetes Jobs API or AWS Batch

Your analysis type, Data Visualization, Exploratory Analysis, Production Analysis, or Machine Learning Pipeline, will inform the infrastructure choices you make on AWS. Each analysis type has it's own distinct requirements.

We'll go over the chart above from left to right. This is only meant to be an overview, and we will go into more detail about each of these later.

You will also notice that there is quite a bit of overlap and grey area between the different analysis types. Here I'll give an overview of each analysis type, and later we'll go over individual case studies that show how these appear in the real world.

6.1 Interactive Data Visualization with RShiny or Dash

The first type is data visualization. I list Dash and RShiny here, because chances are that you're using one of these frameworks or a similar one.

For this discussion we will exclude serverless solutions as those are not generally applicable to visualizing large scale genomics datasets.

Definite requirements include:

- One or more servers (EC2 instances)
- Storage
- Ability to expose ports to web traffic

Simple is rarely the case in Bioinformatics, and we quite frequently need to consider that somewhere down the line we'll need to consider datasets that don't fit into memory, along with potentially scaling out computations to multiple nodes. This is where scalable computational clusters, such as [Dask] or [ApacheSpark] come into play, along with file types that are built for scientific data such as [Zarr] and [ApacheParquet]. Let's create a list of nice to haves.

- **Scale for spiky traffic loads**
  - I say scale here because what you want is for the number of instances you're running to go down during low traffic and up during high traffic with no manual intervention.
- Interface with various microservices such as databases, message queues, Apache Spark clusters or Dask clusters, etc.
- **Ability to track and monitor memory and CPU usage.**
  - This is potentially very important if you're unsure what your memory/cpu requirements are, or if you have an application that can dynamically create different analyses or accept different datasets.

For the first set of requirements you could go with a simple solution, such as [EC2] or [Lightsail]. Each essentially a server and you'll be able to interact with it using SSH.If you're building out a demo, or an application that will receive only a small amount of traffic, then either of these would be a good choice, but leave off our potential requirement for load balancing or easily deploying and interacting with other services.

As a quick side note I truly enjoy the Lightsail experience and don't understand why it hasn't gained more popularity with the scientific community.

If you need to consider scale, or just about anything in that second list in your data visualization application then AWS has several built in solutions for scalable services, such as [Fargate], [Beanstalk], and Managed Elastic Kubernetes, [EKS]. Later we will go more into depth on [EKS] as it's by far my favorite solution.

6.2 Exploratory Analyses

Exploratory analyses fall a bit into their own category. We need compute resources, and we need them *now*. Possibly by last week before we even received the data. We always need storage, and we might need fancier web based programming environments, or we may be satisfied simply having SSH access to a compute node.

Our requirements will include a terminal, preferably a web based terminal for data visualization so we don't have to continuously run rsync, a scalable amount of computational resources (memory, CPU) that we can request in an on demand fashion, and storage. We'd also like an IDE such as [Jupyterhub] and [RStudio].

The best solution I have encountered for this on AWS are HPC clusters build with [AWSParallelCluster]. A very close second is [JupyterhubOnKubernetes]. There are of course, pros and cons to each deployment strategies.

To be honest I think both are very good solutions, and it really depends on personal preference. There is an exception to that, which is that if you are running large scale analyses with multiple steps, such as the GATK Variant Calling Pipeline, that you are probably better off using an HPC solution which is built exactly for that.

We will go more into these solutions in the next chapter.

6.3 Machine Learning Pipelines

Machine learning pipelines and model deployment are much more difficult to pin down. As with most of our solutions there are AWS specific services such as [SageMaker], and more generalized ML frameworks such as [Tensorflow] , [PyTorch] , or [ApacheSpark].

Machine learning pipelines also have several different stages that have different needs.

- Labelling and annotating
- Training
- Testing and Validating
- Model Deployment

Then, of course, it is extremely likely a researcher will go through many iterations of these steps.

For ML Pipelines let's say our requirements are compute servers, storage, software we more than likely don't want to have to compile, install, or configure ourselves, which, to be fair, is the case for all scientific software, along with possibly having a web deployment aspect such as we saw with data visualization.

The labelling and annotation step will likely be some form of a web interface, such as [LabelStudio] (my favorite), or a custom application build on top of [Jupyterhub] or [Dash] . These can be deployed using any of the strategies we discussed in Data Visualization with Dash or RShiny.

The steps after annotation depend on the ML Framework you choose. For most of these you will want general compute infrastructure, either [EC2] or an autoscaling solution such as a HPC cluster built with [AWSParallelCluster] or [EKS] . Then, of course, you may want some level of automation to tie your different steps together. With ML hardware matters quite a bit more than it normally does in Bioinformatics, where we don't tend to take advantage of much beyond multithreading. Many ML frameworks, such as Tensorflow, run much faster on GPU, so that could be an option to consider. Luckily, [AWSParallelCluster] and [EKS] both support GPU nodes.

You may also need very different compute power at different stages in the pipelines. Training may be your memory hog, while evaluating can be as simple as freezing your model and throwing on a [Lightsail] instance wrapped in a REST API. You may also be dealing with large datasets, such as very high resolution images or video, that take some time and computational power, even to evaluate.

6.4 Production and Large Scale Analysis

Large Scale analyses are analyses that are generally not well suited for a single [EC2] instance, but need a more robust solution behind them such as an autoscaling cluster. Here we will discuss which type of autoscaling cluster is best for each analysis type.

6.4.1 Bash Pipelines Analyses

This is the type of analysis Bioinformaticians are most familiar with, and you may already use frameworks such as [Snakemake] or [Nextflow] to manage your pipelines. It involves steps such as those in the [Samtools] workflow, mapping and aligning reads, QC, and variant calling. Each of these steps are accomplished using a command line utility.

Depending on the amount of power needed these may be done on a single [EC2] instance, or an HPC cluster built with [AWSParallelCluster] , along with [EFS] and possibly [S3] file storage.

6.4.2 Code Programmatic Analyses

What if you have a different kind of analysis? Instead of the kind of analysis that a long series of bash commands, such as a variant detection workflow with [GATK], but instead requires waiting on a trigger, such as sequencing data being ready or a row appearing in a database, making a decision or even a series of decisions that are more easily written as code, and then executing some analysis.

This particular type of analysis is not well suited towards HPC, with either [SLURM] or [Batch] . HPC generally assumes you are executing task A -> task B -> task C. You can fine-tune this somewhat by using dependency resolution, but really you are better off using a framework that is more well suited to this type of problem.

For me, [ApacheAirflow] is that system. It is designed with analysis in mind, meaning that at its primary function is to execute a series of tasks, but since these tasks are designed in code it is very flexible.

Most Machine Learning pipelines would be an example of a workflow that requires the flexibility of a code based engine, and there is quite a bit of overlap between ML pipelines and Programmatic and Code Base analyses.

6.5 Production Analyses Frameworks

Production analyses are, in my opinion, analyses that have matured to the point that they are mostly hands off. They should be versioned, preferably in docker containers, and usually have some level of automation to kick them off, along with terrifyingly paranoid levels of logging, error checking, and QC reports. There is probably a large workflow diagram hanging out on somebody's wall somewhere as well.

In the real world these are often marked by a services agreement as well. An example of this would be that you have agreed to run an entire variant calling pipeline on whole exome data, from raw reads to filtered variants, and that data is guaranteed to be delivered within 48 hours. Adding on these additional levels of complexity make it very necessary to have a framework that can run, track, and manage various analyses. My first choice in framework is [ApacheAirflow], but there are of course others such as [Snakemake].

Airflow is where we host the business logic of our workflow, and ideally, not much else. This is where we give instructions such as "run training after annotation". Airflow uses *Operators* to run each node in your workflow, and that makes it completely agnostic as to where you actually run your computation. With Airflow you can train by SSHing over to a server with Tensorflow installed, or use Sagemaker. If your team is already using a Bioinformatics workflow engine such as [Snakemake] you can create an Operator to kick that off as well.

We'll go over a much more indepth pipeline in the *HCS CellProfiler Case Study*.

6.6 IDEs and Analysis Development Environments

For each analysis type we need some manner of IDE and development environment. We will need a place to develop analyses, and to explore data analysis. My favorite method of dealing with a workflow that is so mixed is to use a combination of [ApacheAirflow] and [Jupyterhub], with Jupyterhub usually on my HPC cluster.

We use Jupyterhub as first a platform for exploration. Your data scientists or staff will usually need someplace they can log into and run exploratory analysis, create configurations, and fine tune their pipelines.

Jupyterhub is also extremely handy in that you can create services that are managed within your Jupyterhub cluster. These can be just about anything that can be deployed from a web browser, even [RStudio] for R users, or more typical web applications such as [LabelStudio] for annotating images, documentation services, wikis, or custom [RShiny] and [Dash] applications. Each of these web services can be setup to run alongside [Jupyterhub], or run by individual scientists as a part of their development processes.

CHAPTER

SEVEN

CLUSTER TYPES

In this chapter we'll cover the 2 most common clusters I use on AWS, HPC Clusters built with [AWSParallelCluster] and Elastic Kubernetes Clusters managed with [EKS].

Then, I will once again give the disclaimer that these are not your only cluster solutions. AWS has oodles of solutions all at your AWS console enabled fingertips, and you could plug and play any number of them. These are the solutions I am most fond of and your mileage may vary.

I want to hammer home the point here that no matter the solution, we are spinning up linux servers, and anything you can do with a linux server you can do with an AWS cluster.

Underneath the hood these solutions are very similar. We have nodes that we want to somehow call up programatically so we're not paying for resources we're not using. On an HPC cluster this happens when a user submits a job. On a Kubernetes cluster this is usually kicked off by an application, such as when a user logs into the Jupyterhub portal.

7.1 Elastic High Performance Computing Clusters

One of the most straight forward solutions for exploratory and interactive analyses is to use [AWSParallelCluster] to build out an auto-scaling HPC cluster on AWS.

Mostly you tell it what instance type you want, which will depend on how much memory and how many CPUs you want along with network speed, the minimum and maximum amount of compute nodes to keep available, and optionally you can give it a set of installation instructions that will be applied to any of your nodes when they start up.

Then you say *create my cluster* and poof! HPC cluster at your disposal!

AWS Parallel Cluster builds you out an HPC cluster based on a templated configuration file. There is, of course, a nearly endless amount of customization and configuration you can apply. Once your cluster is provisioned you can do any of the tasks you would normally do, such as set up networked file storage with [EFS], install software, configure services, etc.

7.1.1 Pros

- An HPC environment will feel very natural for many Bioinformatics Users.
- The autoscaling is very straight forward. User submits job, node spins up. Job ends. Node spins down.
- The AWS Parallel Cluster wizard is really nice. Once you're setup administering the cluster is mostly basic Linux systems administration.
- Customizing is straight forward. Create an S3 bucket with your post installation scripts to create users or hook up to a user directory, install packages, etc.
- You can even configure the base AMI that is used on the cluster to include things like docker.

- Since you're essentially dealing with a bunch of remote servers you can all the things you would normally do on remote servers. Anything that can be run on the command line is fair game. This includes IDEs such as [Jupyterhub] or [RStudio], applications such as [LabelStudio]
- The HPC scheduler itself is smart enough to stack users on a node depending on their resource requirements.

7.1.2 Cons

- Before parallel cluster version 2.9.0, the biggest draw back was that you can only have one compute type (t2.medium, t2.large, etc) for your cluster. This is no longer true if you use the SLURM scheduler and enable multiple queues. If you are wondering this was the code change that made me throw my hands up and decide *no code* for this book!
- People who don't like HPC *really don't like HPC*. I somehow manage to talk to a lot of people that tell me HPC is dead. HPC brings about very strong feelings.
- Because the cluster is on demand functionality such as interactive jobs with *srun* don't always work if a node is not already available and in the running state. You can get around this by submitting a job and then ssh over to the node, but not everyone likes that additional step.
- If your users are unfamiliar with a terminal and SSH an HPC system will have a steep learning curve!

7.1.3 IDEs and Development Environments

Some of the cons for HPC *used* to include it is not particularly well suited towards newer interactive data analysis frameworks, such as [Jupyterhub] and [RStudioServer] , that have become increasingly popular in recent years. These problems have very much been solved with various other AWS solutions such as easy Load Balancing and SSL setup. [Jupyterhub] itself supports spawning with an HPC cluster using the Jupyterhub Batch Spawner. The main downside to using [Jupyterhub] on HPC is that it requires a bit more configuration than the [JupyterhubOnKubernetes] solutions, but not much more.

Another con is that it can be a bit more difficult to get started with parallel computing libraries that are *not* MPI, such as [Dask] and [ApacheSpark]. If you understand the fundamentals though you can easily deploy HPC jobs using any of these technologies. Some, such as [Dask] even support HPC with Dask on HPC.

On the pros side people get HPC. Many scientists will have experience with it. It's really a bunch of servers strung together with a scheduler. There's one or more shared file systems, you load modules, you submit jobs, and you get your work done. Whoever is administrating the clusters can add users and adjust filesystems permissions as they would on any Linux server. It's the kind of service you can deploy and it's just there. It's steady, reliable, and not going anywhere anytime soon. I'm also going to count on the plus side that your friendly HPC admin probably doesn't care about whatever shiny new tool showed up on github last week, but if you prefer to live on the bleeding edge, or have devs that do, this may not be a plus for you.

As is typical with our analysis types an exploratory analysis will often fall into a gray area with data visualization. Because really, what is exploring data without making a million charts and graphs?

7.2 Kubernetes

[Kubernetes] is a cluster solution built by Google for container orchestration. It is not specific to AWS, meaning it can run locally on your laptop, within a data center in your universities basement, on AWS, or on another cloud provider.

There are a lot of very techy terms thrown around when discussion Kubernetes. Mostly, it is a set of APIs that sit on top of a bunch of linux servers, or for AWS EC2 instances. Kubernetes supplies us with a number of interfaces that we can use to start, stop and monitor applications and services deployed as docker containers.

These applications can be what you would expect, such as web servers, but also include distributed computing libraries such as [Dask] and [ApacheSpark]. Because the scientific communities have so unexpectedly agreed upon Kubernetes as their platform of choice these solutions are extremely well supported and have all kinds of shininess associated with them, such as monitoring dashboards and autoscaling.

Along with [Kubernetes] is a package manager called [Helm] that bundles up all sorts of deployment scenarios and their configurations. Examples of these include [JupyterhubOnKubernetes] and Dask on Kubernetes.

Like our other solutions, underneath the hood we are dealing with Linux servers. Similar to our HPC clusters we can use networked file storage with [EFS], set environmental variables, install packages, etc.

If you have an application that has a lot of moving pieces and there is a [Helm] chart for it Kubernetes is probably your best bet.

7.2.1 Pros

When you get the hang of it Kubernetes is a *nice* way of deploying web applications. I've reached the point where I only support Kubernetes for large scale web apps in my consulting because the tooling just can't be beat.

- Kubernetes has been extremely well accepted by the scientific computing community. It's so difficult to get any group of developers to accept or band together on any one framework that this is a much larger perk than can be expressed with mere words.
- The Helm package manager packages up not only the services themselves, but also takes care of a great deal of the configuration involved.
- I can't talk about Helm without talking about the Bitnami Helm Charts. I've been using Bitnami to deploy applications for longer than I am willing to admit and they are awesome.
- Have I mentioned the support of the scientific community enough? There are a wide array of tools and libraries available all preconfigured with [Helm] and ready to go, including [Dask], [ApacheSpark], and [JupyterhubOnKubernetes].
- Kubernetes can be deployed on your laptop, in a local datacenter, or in the cloud.
- There are prototyping solutions, such as [Minikube] to help you learn Kubernetes and test applications without deploying an entire cluster.
- Kubernetes clusters can be deployed with the click of a button on both AWS and GCP.

7.2.2 Cons

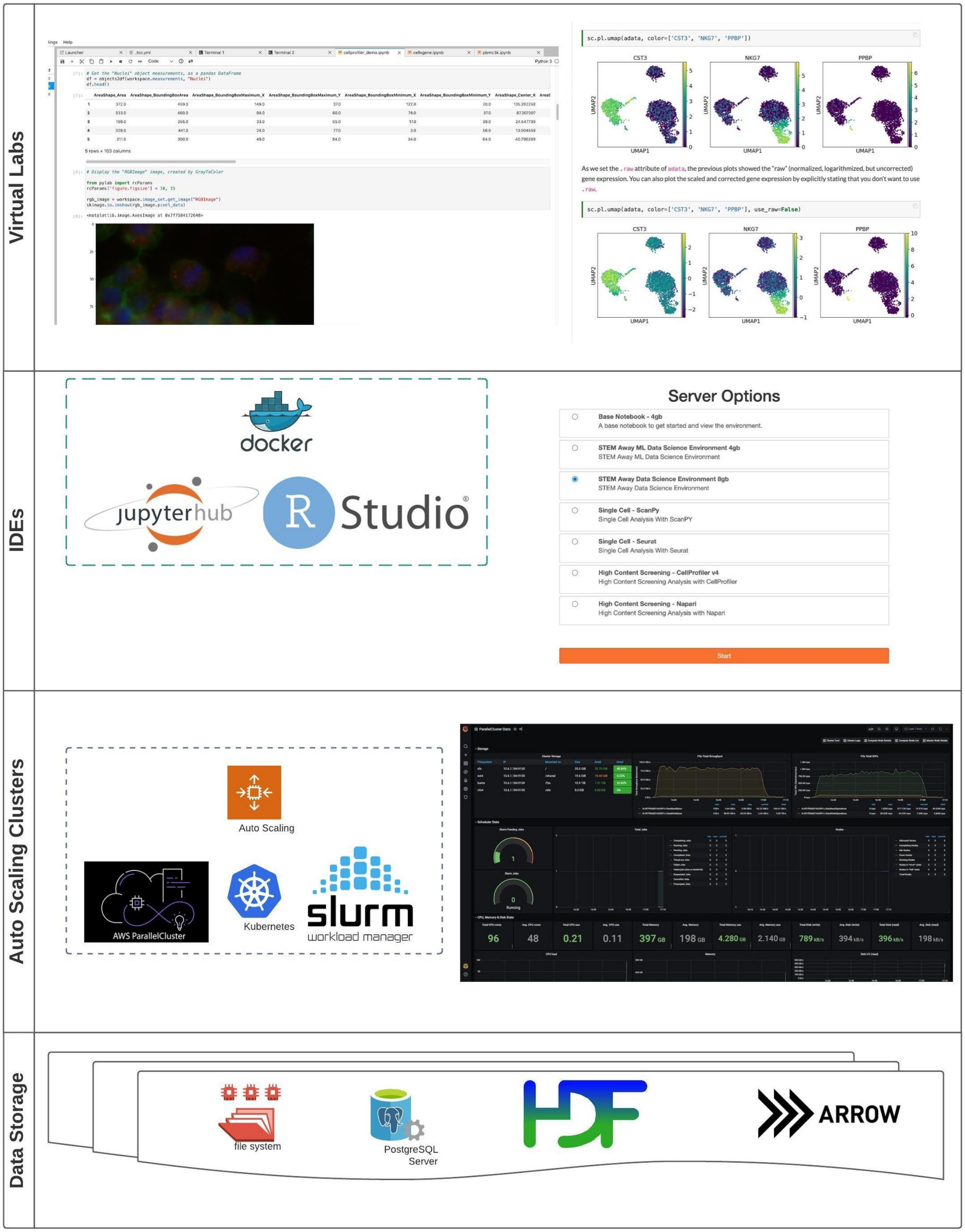
- Kubernetes is a full on DevOps framework that has a rather steep learning curve. It requires knowledge of docker, and might be difficult to learn if you don't have any prior experience packaging and deploying applications.
- Sometimes your engineers get so fed up with everything that they swing to an opposite extreme where they will only use bash and ssh and everyone can *just deal*.

CHAPTER

EIGHT

WE HAVE OUR CLUSTERS! NOW WHAT?

Now that we have our clusters it's time to discuss what we can deploy on them. As always, our aim here is to create an infrastructure that aids in *good science* in general and **great Bioinformatics** in particular!



```
[7]: # Get the "Nuclei" object measurements, as a pandas DataFrame
     df = objects2df(workspace.measurements, "Nuclei")
     df.head()
```

	AreaShape_Area	AreaShape_BoundingBoxArea	AreaShape_BoundingBoxMaximum_X	AreaShape_BoundingBoxMaximum_Y	AreaShape_BoundingBoxMinimum_X	AreaShape_BoundingBoxMinimum_Y	AreaShape_Center_X	AreaS
1	372.0	459.0	149.0	37.0	122.0	20.0	135.282258	
2	333.0	460.0	98.0	60.0	78.0	37.0	87.207207	
3	199.0	256.0	33.0	55.0	17.0	39.0	24.647739	
4	329.0	441.0	24.0	77.0	3.0	56.0	13.604659	
5	211.0	300.0	49.0	84.0	34.0	64.0	40.796209	

5 rows × 103 columns

```
[8]: # Display the "RGBImage" image, created by GrayToColor

     from pylab import rcParams
     rcParams['figure.figsize'] = 10, 15

     rgb_image = workspace.image_set.get_image("RGBImage")
     skimage.io.imshow(rgb_image.pixel_data)

[8]: <matplotlib.image.AxesImage at 0x7f7584172640>
```

```
sc.pl.umap(adata, color=['CST3', 'NKG7', 'PPBP'])
```

As we set the .raw attribute of adata, the previous plots showed the "raw" (normalized, logarithmized, but uncorrected) gene expression. You can also plot the scaled and corrected gene expression by explicitly stating that you don't want to use .raw.

```
sc.pl.umap(adata, color=['CST3', 'NKG7', 'PPBP'], use_raw=False)
```

Server Options

○ Base Notebook - 4gb
A base notebook to get started and view the environment.

○ STEM Away ML Data Science Environment 4gb
STEM Away ML Data Science Environment

● STEM Away Data Science Environment 8gb
STEM Away Data Science Environment

○ Single Cell - ScanPy
Single Cell Analysis With ScanPY

○ Single Cell - Seurat
Single Cell Analysis With Seurat

○ High Content Screening - CellProfiler v4
High Content Screening Analysis with CellProfiler

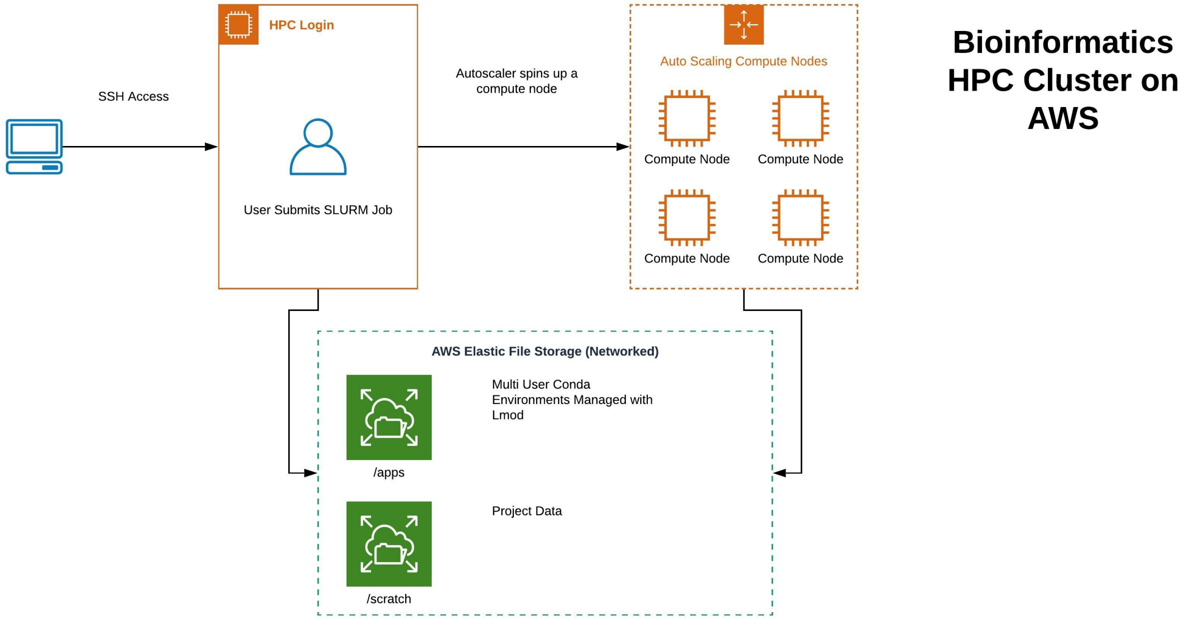
○ High Content Screening - Napari
High Content Screening Analysis with Napari

Start

8.1 Bioinformatics Solutions Ecosystems

We will discuss a few Bioinformatics Analyses Ecosystems, mainly SLURM with HPC and [AWSParallelCluster] and [Jupyterhub] with [Kubernetes]. Both are excellent solutions for creating a managed data science environment for your lab, institute, or startup, but have slightly different use cases we will go over below.

8.1.1 SLURM on AWS


SSH Access
HPC Login
User Submits SLURM Job
Autoscaler spins up a compute node
Auto Scaling Compute Nodes
Compute Node
Compute Node
Compute Node
Compute Node
Bioinformatics HPC Cluster on AWS
AWS Elastic File Storage (Networked)
/apps
Multi User Conda Environments Managed with Lmod
/scratch
Project Data

Traditionally, many researchers run their analyses on an HPC environment. This has several perks, most notably that you can leverage the power of having multiple computers all talking to one another and actually get your analysis run this century. If you haven't used HPC there are a lot of benefits. You can easily set up a multiuser environment, and you can transparently spin instances up and down by submitting jobs, with no manual intervention from the user, in order to scale analyses.

You can deploy several flavors of HPC cluster, including your typical SLURM and PBS or some of the new, potentially shinier, AWS solutions such as AWS Batch, using the AWS ParallelCluster service.

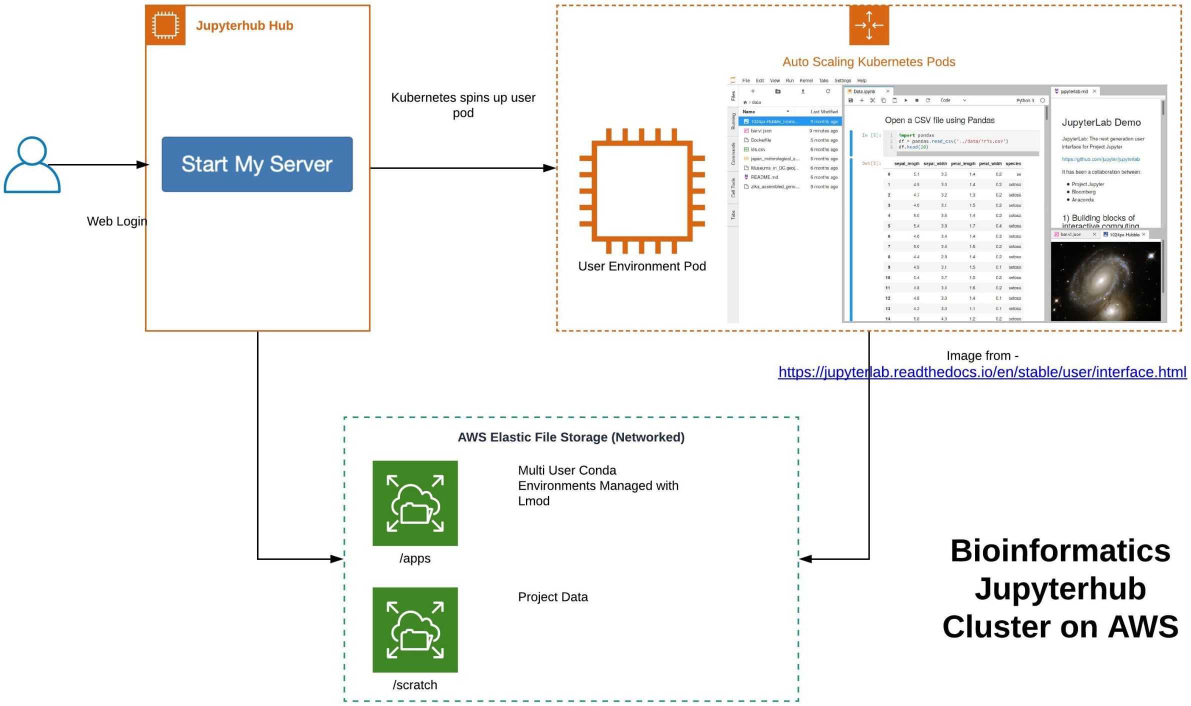
Funny storytime! AWS and GCP both released their own HPC-esque solutions on their respective platforms. These were embraced and dismissed by the scientific community in just about equal proportions. Several teams responded back with "well that's nice, but we just want our HPC", and StarCluster was born! Then at some point in time, AWS ParallelCluster came into being, which is a wrapper deploying both traditional HPC environments and also making newer solutions such as AWS Batch look like HPC to make the "you kids need to get off my lawn and give me HPC" scientists happy.

Now we are back to our regularly scheduled actionable deployment strategies.

I use an infrastructure as code solution such as [Terraform] or [CloudFormation] in order to deploy other resources. Most often these are networked file systems such as [EFS], AWS RDS databases for logging with Redash or project analysis storage, [S3] buckets to host startup scripts, user data and archives, security groups for access control, and/or additional [Dask] or [ApacheSpark] clusters for computation. For the Dask and Spark clusters I give lots of disclaimers that these solutions work better on [Kubernetes] and try to steer everyone towards a hybrid approach.

From there, it's time to actually get these on the cluster, which is done by using the startup scripts hook in the ParallelCluster. I nearly always install Modules, and set up a base Miniconda3 environment to allow users to bootstrap their own environments. I also set up a base [JupyterHub] and [RStudio] Server installation as these are always popular requests on SLURM clusters that are used for Bioinformatics analyses.

8.1.2 Jupyterhub on Kubernetes


Jupyterhub Hub
Start My Server
Web Login
Kubernetes spins up user pod
Auto Scaling Kubernetes Pods
User Environment Pod
Open a CSV file using Pandas
JupyterLab Demo
Image from - https://jupyterlab.readthedocs.io/en/stable/user/interface.html
AWS Elastic File Storage (Networked)
Multi User Conda Environments Managed with Lmod
/apps
Project Data
/scratch
Bioinformatics Jupyterhub Cluster on AWS

Using [JupyterhubOnKubernetes] is a fantastic solution for data science teams that are already invested in the scientific python ecosystem.

I truly can't say enough great things about this solution. It is extremely customizable, and the implementation is very well thought out.

From a birds eye view it looks quite similar to the HPC solution, except that you're giving your users Jupyterhub or Jupyterlab notebooks along with a remote server. The autoscaling looks about the same, except its user logins Jupyterhub, requests a server, node spins up, user logs in, logs out or is idle, node spins down.

Generally, with the HPC solution I create an EFS for the apps, and have a centrally managed software repository with mostly conda environments. You can still do this, because you can use networked file storage *literally anywhere*, including on Kubernetes with the efs provisioner helm chart. You also have the option of giving your users preconfigured environments with docker containers. If your team heavily uses docker containers you may prefer this solution.

Jupyterhub also has lots of nice wrappers for almost anything you would want to do through a browser. You can deploy separate proxied services for anything you run in a browser, including services for documentation,to run RStudio Server, or to launch your Computer Vision Image Labelling Infrastructure..

If you or your team builds out data visualization internal solutions with [RShiny] or [Dash] you can set these up as services within Jupyterhub. This gives you a single authentication strategy to enable an endless possibility of services.

You don't need to choose just one cluster type. If you need the power of an HPC scheduler to submit multistep, long running pipeline jobs along with your super shiny looking notebooks your HPC Cluster and Jupyterhub Cluster solutions can coexist using SSH with the remote ikernel python package. (Side note: I really love that package. It's *gold* and I use it all the time.)

Depending on the computation you're doing you may be able to side step using an HPC by giving your users access to auto scaling Dask Clusters. This is becoming an ever more popular approach as scientific software.

Pros

- You can configure your cluster as a central hub for all of your data visualization and web solutions, all behind a single authentication wall.
- This solution is much more natural for users that are already invested in scientific python solutions.
- Jupyterhub Managed Services! Do you run it through the web? You can run it as a part of your Jupyterhub Cluster.
- Many authentication mechanisms are supported.
- You still get a terminal, and can interact with any of your other services and clusters through SSH.
- It's built on Kubernetes, with depending on your view of Kubernetes is either a pro or a con. ;-)

Cons

- See my previous comments about Kubernetes.
- If your users don't like Jupyterhub they might be a bit cranky, but they can still get a terminal so they'll probably calm down.
- You can use your HPC cluster to launch notebooks, but can't use Jupyterhub to deploy long running jobs that are common to Bioinformatics.

If I had had to give a one sentence summary I would say an HPC cluster is considerably better for large analyses, and that a Kubernetes cluster is better for deploying applications, but also has many utilities within the data science ecosystem.

8.2 Apache Airflow For Production Analyses on Kubernetes

Apache Airflow is a wonderful and amazing framework, but is an absolute beast to try and install manually. I wouldn't even really bother trying to get anything but the most basic of demo environments up and running without Docker, and for production environments I only support Apache Airflow on Kubernetes.

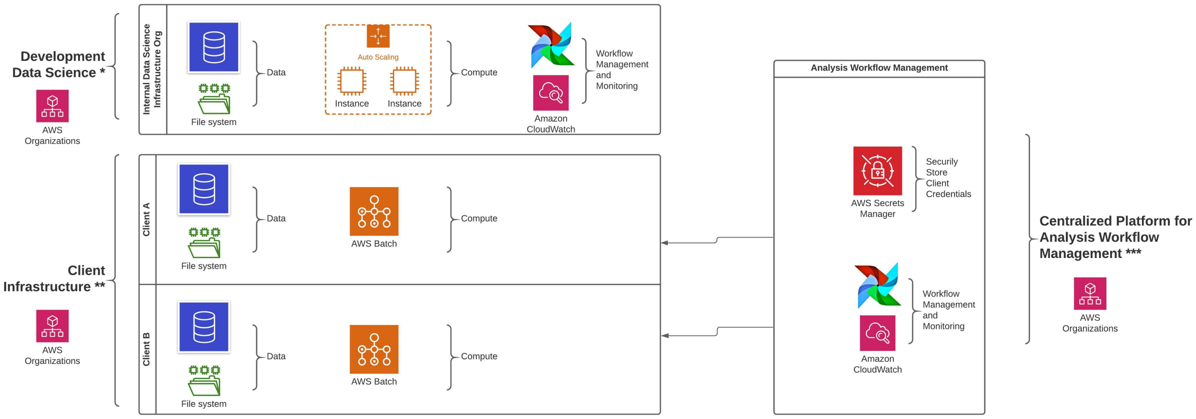
My favorite way to get up and running with an Airflow development environment is to use the Airflow Bitnami Docker Compose Stack. The development environment can be run anywhere Docker is installed.

In order to get going in production I exclusively use the Airflow Bitnami Helm Chart for Kubernetes.

[ApacheAirflow] is, in my opinion, the best solution if you need HIPAA compliance for your data. Because it is run with Kubernetes each analysis is isolated within a pod. Secrets, such as database credentials, AWS acccess keys, or other types of secret data can be stored using either Kubernetes Secrets or the AWS SSM to securely store these secrets within Airflow itself to use in analyses. This uses the AWS Secrets Manager , which is itself HIPAA compliant.

I particularly like Airflow for startups or labs that have an analysis as a service model. This means they get, from somewhere, usually as a result of personalized medicine initiative at a hospital. Once the data is recieved, either on an [S3] bucket or an FTP file system baccked by [EFS], an Airflow Sensor is pinged that there is new data and kicks off an analysis.

One of the major benefits to Airflow is that it is completely agnostic to where your analysis actually runs. That means Airflow can act as a spider in the web hanging out on a Kubernetes cluster. Depending on the amount of separation you need each analysis could be running on a dedicated client infrastructure, complete with it's own HPC cluster, or a single analysis cluster that is separated by docker containers or Kubernetes Pods. The level of separation you need will very much depend upon your data and your needs, but because of the way Airflow is organized many configurations are possible.



* Internal Data Science Infrastructure
This is the place where your data scientists work on developing new analyses, protocols, reporting methods, etc.

** Client Infrastructure
For data compliance normally each client has their own data in their own organization. This may include databases, filesystems, and compute clusters.
Having each client in their own AWS organization also makes book keeping and billing much simpler.

*** Centralized Platform for Analysis Workflow Management
Workflows and analyses can be centrally managed using Apache Airflow, with client credentials such as AWS Access Keys, database authentication, SSH keys, etc securely stored using AWS Secret Management.

In this way analyses and metadata are managed using a central system that can speak to other systems (JIRA, Slack, etc). Analyses can be versioned and testsed as each analysis is code, and software environments can be stored in docker images.

CHAPTER

NINE

CASE STUDY - HIGH CONTENT SCREENING WITH CELLPROFILER ON AWS

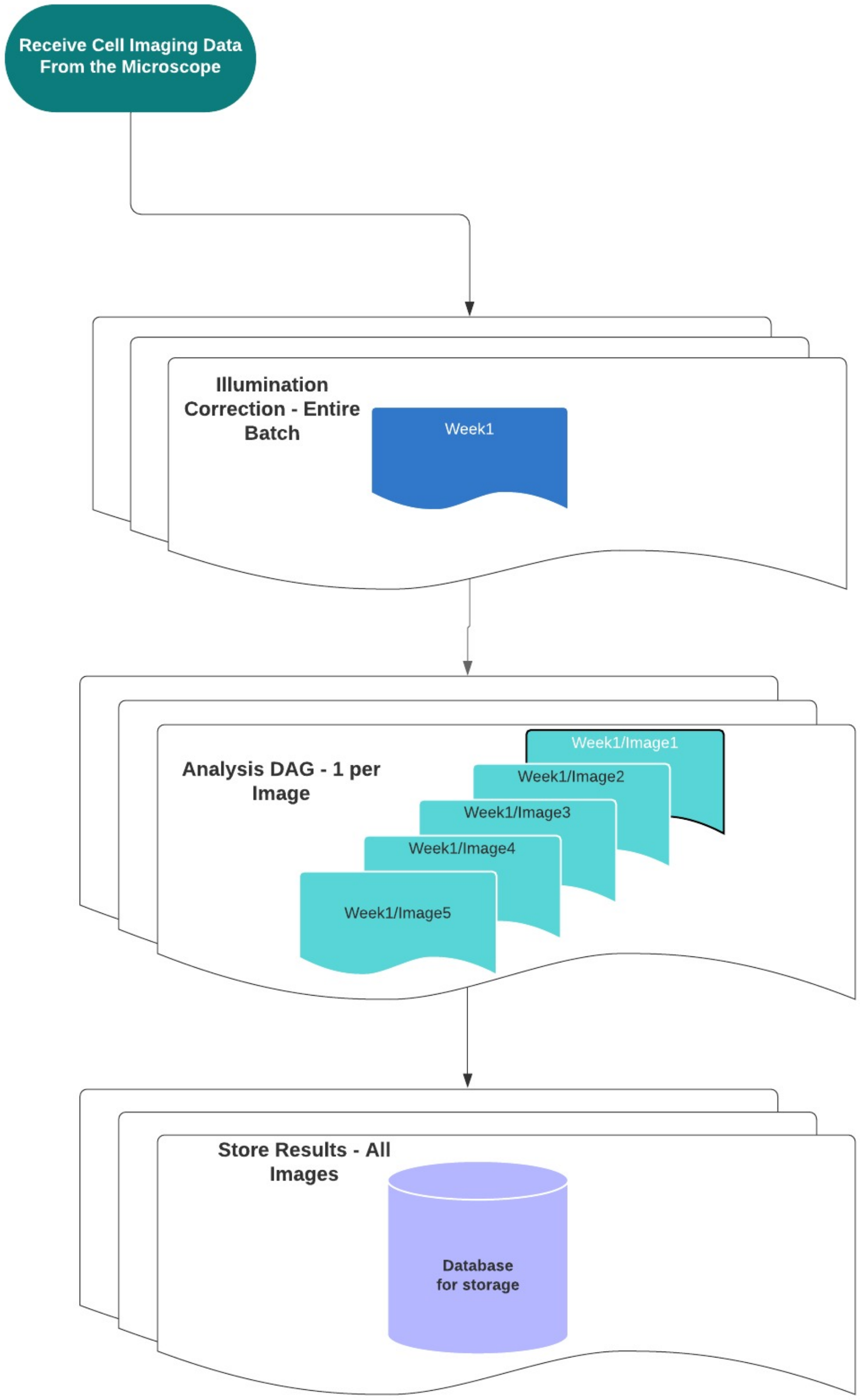
If you are running a High Content Screening Pipeline you probably have a lot of moving pieces. As a non exhaustive list you need to:

- Trigger CellProfiler Analyses, either from a LIMS system, by watching a filesystem, or some other process.
- Keep track of dependencies of CellProfiler Analyses - first run an illumination correction and then your analysis.
- If you have a large dataset and you want to get it analyzed sometime this century you need to split your analysis, run, and then gather the results.
- Once you have results you need to decide on a method of organization. You need to put your data in a database and set up in depth analysis pipelines.

These tasks are much easier to accomplish when you have a system or framework that is built for scientific workflows.

In this we're going to discuss the BBBC021 dataset and how I would organize and batch the analysis. This book is all about AWS, so we will also discuss how to leverage AWS in order to automatically scale our compute infrastructure while running this analysis.

BBBC021 is a fairly typical HCS analysis. Data comes in. We need to do a correction across the batch for illumination. Then, we'll process each image individually using a provided Cellprofiler analysis. This isn't included in the code, but typically once there are results, typically one per image, we would want to do something such as aggregate those results to a database, backup the CSVs, upload to a data lake solution, etc.


Receive Cell Imaging Data From the Microscope
Illumination Correction - Entire Batch
Week1
Analysis DAG - 1 per Image
Week1/Image1
Week1/Image2
Week1/Image3
Week1/Image4
Week1/Image5
Store Results - All Images
Database for storage

This case study uses Docker and Docker Compose for preconfigured software stacks. Even if you don't know Docker I have included all the commands for you to run the analyses.

9.1 Enter Apache Airflow

[ApacheAirflow] is:

> Airflow is a platform created by the community to programmatically author, schedule and monitor workflows.
>
> *Apache Airflow <https://airflow.apache.org/>*

There are a ton of great introductory resources out there on Apache Airflow, but I will very briefly go over it here.

Apache Airflow gives you a framework to organize your analyses into DAGs, or Directed Acyclic Graphs. If you aren't familiar with this term it's really just a way of saying Step3 depends upon Step2 which depends upon Step1, or *Step1 -> Step2 -> Step3*.

Apache Airflow uses **DAGs**, which are the bucket you throw you analysis in. Your DAG is comprised of Operators and Sensors. **Operators** are an abstraction on the kind of task you are completing. These will often be Bash, Python, SSH, but can also be even cooler things like Docker, Kubernetes, AWS Batch, AWS ECS, Database Operations, file pushers, and more. Then there are also **Sensors**, which are nice and shiny ways of waiting for various operations, whether that is waiting on a file to appear, a record in a database to appear, or another task to complete.

Out of the box you get lots of niceness, including a nice web interface with a visual browser of your tasks, a scheduler, configurable parallelism, logging, watchers and any number of executors. As all of your configuration is written in code it is also extremely flexible. It can integrate with existing systems or stand on it's own.

There are any number of scientific workflow managers out there, and by the time I finish this chapter a few more will have popped into existence. Apache Airflow is my favorite, but you should shop around to see what clicks with you!

Before we get into our analysis I want to talk a bit more in depth about [ApacheAirflow].

9.1.1 DAGs and Tasks

DAGs are directed acyclic graphs. You may have seen these before as decision trees or just directed graphs. If you're coming from a Bioinformatics background you've probably seen these referred to as workflows or pipelines, which contain the business logic for running your analysis on one or more computers.

DAGs are the overarching design of your analysis, what gets executed and in what order. A DAG is comprised of many tasks, and tasks can be shared among DAGs.

9.1.2 Operators

The operators are the workhorses of the Apache Airflow world. This is where the magic happens, where the numbers are crunched, where the data is split, etc. These operators are also completely agnostic to your run platform. Operators have built in logic to execute code locally with Bash or Python, remotely with SSH, in containerized instances such as Docker or on a Kubernetes cluster, or through AWS services such as [Sagemaker] , making Airflow the perfect fit for the hybrid solutions we will get into in the case studies chapter.

9.1.3 Sensors

Sensors are a particular kind of operator that wait on some condition to trigger their portion of the DAG. This may be when a file appears on a filesystem, a row appears in a database, some time constraint, or anything you can think of to wait for.

Airflow has any number of what they call operators, which are various ways of getting stuff done. Maybe you have a decision tree you need to execute. You can do this with a Branching Operator. You may also want to execute some code on a remote cluster using the Kubernetes Operator, or crunch some numbers, while taking advantage of an Apache Spark or Dask cluster, using the Python Operator.

You can also create custom operators by simply extending any operator or sensor class. I find that these are an excellent way to keep my code organized and reusable.

Additionally, since Apache Airflow is built to be a workflow orchestrator it has built-in functionality for logging and alerts, as well as hooks for each task such as on start, on finish, on success, on fail. You can use this kind of functionality for generating reports, programmatically updating project management software such as JIRA, or to create status alerts and updates over email or slack.

Apache Airflow is an application, and you can deploy it on Kubernetes using AWS [EKS] . You get all the same bells and whistles you get deploying any application on Kubernetes, including logging infrastructure, CPU and memory profiling, and the ability to scale from one to many instances. My favorite way to deploy Airflow is to use the Bitnami Airflow Helm Chart.

9.1.4 Computational Backends

I briefly touched on this earlier, but one of the perks that initially drew me to Apache Airflow is just how completely agnostic it is to your compute environment. You could have a laptop, a single server, a HPC cluster, or execute on the AWS or GCP. Airflow itself does not care. All you need to do is to map out your logic, make sure the data is available, and use whichever operator is appropriate.

9.2 AWS Infrastructure

I nearly always separate out my development infrastructure from my production infrastructure. I really shouldn't be left on an HPC system without adult supervision, and I tend to clobber things frequently. If this is a problem you or your devs have, you may want to follow this advice too.

If you're just getting started with cloud computing don't worry. The case study shown here mostly emphasizes the analysis. Infrastructure should be done in such a way that it is the wizard behind the curtain, and not in your way. My hope is that you dig into this case study, and really think about how you would structure your analysis. Once you know that mapping your analysis to an infrastructure is much more straightforward.

9.2.1 Development

Typically, for development I will develop on a single EC2 instance, or if I have an HPC cluster with [Jupyterhub] handy I will develop on that. In either case I use the Airflow Bitnami docker compose stack.

Airflow doesn't really care where you run your analyses, and this logic is encapsulated in the Airflow operators. From a practical standpoint what this means is that while I am in development I will often use the Docker Operator, and just run the whole thing as a part of a single docker compose dev stack.

9.2.2 Production

For production I like to deploy the Airflow application itself to [Kubernetes] on AWS with the [EKS] service. I have found that this is the best and most fault tolerant way to deploy applications on AWS.

> The analysis logic will stay exactly the same as we saw in development, but I will move from a Docker Operator to an AWSBatch Operator. [AWSBatch] is an AWS HPCesque cluster that has excellent integration with Airflow. Because we're developing the entire time with Docker, switching between our development and production environments is usually seamless.

9.3 Example CellProfiler Analysis Workflow

In this post I'm going to discuss the BBBC021 dataset and how I would organize and batch the analysis.

I decided to go for a simple setup, which is to use Apache Airflow with docker-compose and use the Docker operator to execute the CellProfiler analysis. Once you have your logic and workflow mapped out you could use any operator for any compute infrastructure, whether that is AWS ECS, or an HPC. My favorite lately has been Kubernetes, because it is not tied to any platform and can be used on AWS, GCP or in house. You can use Kubernetes to deploy your data visualization applications such as RShiny, Dash or Redash, and if you are using networked storage or S3 all your applications can access the same data!

9.4 Project setup

Let's setup our project directory structure!

```
mkdir CellProfiler-Apache-Airflow
cd CellProfiler-Apache-Airflow
mkdir -p data
mkdir -p dags
mkdir -p data/BBBC021/Week1
cd data/BBBC021/Week1
wget https://data.broadinstitute.org/bbbc/BBBC021/BBBC021_v1_images_Week1_22123.zip
find $(pwd) -name "*zip" | xargs -I {} unzip {}
# Clean up the zips, we don't need them anymore
find $(pwd) -name "*zip" | xargs -I {} rm -rf {}
cd ../
# Run $(pwd) to check where you are. You should be in /project/BBBC021
wget https://data.broadinstitute.org/bbbc/BBBC021/BBBC021_v1_image.csv
wget https://data.broadinstitute.org/bbbc/BBBC021/BBBC021_v1_compound.csv
wget https://data.broadinstitute.org/bbbc/BBBC021/BBBC021_v1_moa.csv
wget https://data.broadinstitute.org/bbbc/BBBC021/analysis.cppipe
wget https://data.broadinstitute.org/bbbc/BBBC021/illum.cppipe

# Let's create a data file ONLY for the week1 images, the first dataset
head -n 1 BBBC021_v1_image.csv > images_week1.csv
cat BBBC021_v1_image.csv | grep Week1_22123 >> images_week1.csv
```

This is mostly the Apache Airflow configuration from Bitnami. Bitnami is awesome and I use their configurations and images all the time. I made a few modifications to this one to bind our analysis dags, and also made a quick change so we could use the docker operator.

9.5 Docker Image

Generally speaking, I like to keep my Airflow images very clean. Airflow should be the spider sitting in the web. I always prefer to package up my analyses into Docker images, and this analysis is no exception. Luckily for us I already have a CellProfiler image available through BioHub. As a quick side note each BioHub Docker image comes with [Jupyterhub], [RStudio] server, and come prepackaged with much of the [PyData] ecosystem to make getting going as simple as possible. Each of these images can be used standalone, run through on an HPC if Docker is enabled, or as apart of a [JupyterhubOnKubernetes] cluster.

Make sure you are in your project directory. If you are ever unsure of which directory you are in, here are a few terminal commands to help you out.

```
# list all files in the current directory
ls -lah
# list my current working directory
pwd
```

Now, let's run our CellProfiler image with our data and just briefly check it out.

```
docker run -it -p 8888:8888 -v "$(pwd)/data:/home/jovyan/data" \
    dabbleofdevops/cellprofiler-notebook bash -c "jupyter notebook --port 8888 --ip 0.
↪0.0.0"
```

You should see some output that looks like:

```
(base)   Bioinformatics-Solutions-On-AWS git:(master)      docker run -it -p␣
↪8890:8888 -v "$(pwd)/data:/home/jovyan/data" dabbleofdevops/cellprofiler-notebook␣
↪bash -c "jupyter notebook --port 8888 --ip 0.0.0.0"
[I 06:27:56.228 NotebookApp] [nb_conda_kernels] enabled, 1 kernels found
[I 06:27:56.321 NotebookApp] Writing notebook server cookie secret to /home/jovyan/.
↪local/share/jupyter/runtime/notebook_cookie_secret
[I 06:28:00.539 NotebookApp] JupyterLab extension loaded from /srv/conda/envs/
↪notebook/lib/python3.8/site-packages/jupyterlab
[I 06:28:00.540 NotebookApp] JupyterLab application directory is /srv/conda/envs/
↪notebook/share/jupyter/lab
[I 06:28:00.760 NotebookApp] [Jupytext Server Extension] Deriving a␣
↪JupytextContentsManager from LargeFileManager
[I 06:28:01.020 NotebookApp] Serving notebooks from local directory: /home/jovyan
[I 06:28:01.020 NotebookApp] Jupyter Notebook 6.2.0 is running at:
[I 06:28:01.020 NotebookApp] http://b85acbd716e1:8888/?
↪token=f12cb022b8eefb6a2d2d26040c8f8e5a9a7a558bde5e9f52
[I 06:28:01.020 NotebookApp]  or http://127.0.0.1:8888/?
↪token=f12cb022b8eefb6a2d2d26040c8f8e5a9a7a558bde5e9f52
[I 06:28:01.020 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
[W 06:28:01.027 NotebookApp] No web browser found: could not locate runnable browser.
[C 06:28:01.028 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/jovyan/.local/share/jupyter/runtime/nbserver-1-open.html
    Or copy and paste one of these URLs:
        http://b85acbd716e1:8888/?
↪token=f12cb022b8eefb6a2d2d26040c8f8e5a9a7a558bde5e9f52
     or http://127.0.0.1:8888/?token=f12cb022b8eefb6a2d2d26040c8f8e5a9a7a558bde5e9f52
```

Grab the URL at the very end, put it in your browser, and Jupyterhub will pop right up. If you prefer the lab interface change the URL from /tree to /lab.

9.5.1 Grab the Docker Compose Configuration

Grab this file and save in your project root as *docker-compose.yml*. This configuration is based off of the Bitnami *docker-compose.yml* file with a few modifications.

```yaml
version: '2'

services:
  postgresql:
    image: 'bitnami/postgresql:10'
      - 'postgresql_data:/bitnami/postgresql'
    environment:
      - POSTGRESQL_DATABASE=bitnami_airflow
      - POSTGRESQL_USERNAME=bn_airflow
      - POSTGRESQL_PASSWORD=bitnami1
      - ALLOW_EMPTY_PASSWORD=yes
  redis:
    image: bitnami/redis:6.0
    volumes:
      - 'redis_data:/bitnami'
    environment:
      - ALLOW_EMPTY_PASSWORD=yes
  airflow-scheduler:
    image: bitnami/airflow-scheduler:2
    environment:
      - AIRFLOW_LOAD_EXAMPLES=no
      - AIRFLOW_DATABASE_NAME=bitnami_airflow
      - AIRFLOW_DATABASE_USERNAME=bn_airflow
      - AIRFLOW_DATABASE_PASSWORD=bitnami1
      - AIRFLOW_EXECUTOR=CeleryExecutor
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./data:/data
      - ./dags:/opt/bitnami/airflow/dags
      - airflow_scheduler_data:/bitnami
  airflow-worker:
    image: bitnami/airflow-worker:2
    environment:
      - AIRFLOW_LOAD_EXAMPLES=no
      - AIRFLOW_DATABASE_NAME=bitnami_airflow
      - AIRFLOW_DATABASE_USERNAME=bn_airflow
      - AIRFLOW_DATABASE_PASSWORD=bitnami1
      - AIRFLOW_EXECUTOR=CeleryExecutor
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./data:/data
      - ./dags:/opt/bitnami/airflow/dags
      - airflow_worker_data:/bitnami
  airflow:
    image: bitnami/airflow:2
    environment:
      - AIRFLOW_LOAD_EXAMPLES=no
      - AIRFLOW_DATABASE_NAME=bitnami_airflow
      - AIRFLOW_DATABASE_USERNAME=bn_airflow
      - AIRFLOW_DATABASE_PASSWORD=bitnami1
      - AIRFLOW_EXECUTOR=CeleryExecutor
    ports:
      - '8080:8080'
```

(continues on next page)

(continued from previous page)

```
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./data:/data
      - ./dags:/opt/bitnami/airflow/dags
      - airflow_data:/bitnami
volumes:
  airflow_scheduler_data:
    driver: local
  airflow_worker_data:
    driver: local
  airflow_data:
    driver: local
  postgresql_data:
    driver: local
  redis_data:
    driver: local
```

If you're not used to containers this can get a little tricky, but one thing to note is that paths on the host are not necessarily the same as paths in the docker container, for example, our data directory could be anywhere on our host system, but is bound as */data* on our container.

Put the *docker-compose.yml* file in your project directory and bring it up with *docker-compose up*. It may take some time to initialize. This is when I go make tea. ;-)

Once it's up you'll be able to access your Airflow instance at *localhost:8080* with the default configuration.

```
AIRFLOW_USERNAME: Airflow application username. Default: user
AIRFLOW_PASSWORD: Airflow application password. Default: bitnami
```

There won't be anything interesting here yet, because we don't have our analysis in place.

9.6 Grab the CellProfiler Analysis DAGs

First grab the illum dag. Place it in your *dags* folder. It can be named anything, what Airflow references is the *dag_id*, but I'll reference it as *cellprofiler-illum-dag.py*.

```
# dags/cellprofiler-illum-dag.py
from airflow import DAG
from datetime import datetime, timedelta
import string
import random
from airflow.utils import timezone
from airflow.operators.dagrun_operator import TriggerDagRunOperator
from airflow.operators.bash_operator import BashOperator
from airflow.operators.python_operator import PythonOperator
# Depending on which version of airflow you are on you will use either operators.
↪docker_operator or providers.docker
from airflow.operators.docker_operator import DockerOperator
# from airflow.providers.docker.operators.docker import DockerOperator
from airflow.api.common.experimental.trigger_dag import trigger_dag
from airflow.sensors.external_task_sensor import ExternalTaskSensor
from airflow.api.common.experimental import check_and_get_dag, check_and_get_dagrun
import time
import os
from pprint import pprint
```

(continues on next page)

(continued from previous page)

```
from airflow.utils.state import State

this_env = os.environ.copy()

this_dir = os.path.dirname(os.path.realpath(__file__))

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2019, 1, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG('cellprofiler_illumination', default_args=default_args, schedule_
↪interval=None)

EXAMPLE_TRIGGER = """
{
    "illum_pipeline" : "/data/BBBC021/illum.cppipe",
    "analysis_pipeline" : "/data/BBBC021/analysis.cppipe",
    "pipeline" : "/data/BBBC021/illum.cppipe",
    "output": "/data/BBBC021/Week1/Week1_22123",
    "input": "/data/BBBC021/Week1/Week1_22123",
    "data_file": "/data/BBBC021/images_week1.csv"
}
"""

# Volumes are from the HOST MACHINE
illum = DockerOperator(
    dag=dag,
    task_id='illum',
    retries=1,
    volumes=[
        # UPDATE THIS to your path!
        '/path-to-project-on-HOST/data:/data'
    ],
    working_dir='/data/BBBC021',
    tty=True,
    image='cellprofiler',
    command=[
        "bash", "-c",
        """cellprofiler --run --run-headless \
            -p {{ dag_run.conf['illum_pipeline'] }}  \
            -o {{ dag_run.conf['output'] }}  \
            -i {{ dag_run.conf['input'] }}  \
            -data-file {{ dag_run.conf['data_file'] }} \
            -c -r"""
    ]
)


def get_number_of_tasks(data_file):
    """
```

(continues on next page)

(continued from previous page)

```
    Parse the file to get the number of lines
    The number of lines, minus 1 for the header
    is the number of groups
    :param data_file:
    :return:
    """
    file = open(data_file, "r")
    number_of_lines = 0
    for line in file:
        number_of_lines += 1
    file.close()
    return number_of_lines - 1


def watch_task(triggers):
    """
    This is only here for demonstration purposes
    to show how you could dynamically watch the cellprofiler analysis DAG
    :param triggers:
    :return:
    """
    print('-------------------------------------------')
    print('Checking up on our dag...')
    check_dag = check_and_get_dag(dag_id='cellprofiler_analysis')
    dag_run = check_and_get_dagrun(check_dag, triggers[0].execution_date)
    state = dag_run.get_state()
    finished = State.finished()
    unfinished = State.unfinished()

    while state in unfinished:
        time.sleep(10)
        state = dag_run.get_state()

    print('-------------------------------------------')
    print('Dag run finished or dead')
    pprint(dag_run.get_state())


def trigger_analysis(ds, **kwargs):
    """
    Trigger the cellprofiler analysis DAG
    We want one DAG run per row in the datafile, or -f / -l combo
    :param ds:
    :param kwargs:
    :return:
    """
    print('-------------------------------------------')
    print("Here's the conf!")
    pprint(kwargs['dag_run'].conf)
    output = kwargs['dag_run'].conf['output']
    data_file = kwargs['dag_run'].conf['data_file']
    no_tasks = get_number_of_tasks(str(data_file))
    triggers = []
    print('-------------------------------------------')
    print('Triggering our dag...')
    for index, value in enumerate(range(1, no_tasks + 1)):
        trigger = trigger_dag(
```

(continues on next page)

(continued from previous page)

```
            dag_id="cellprofiler_analysis",
            replace_microseconds=False,
            run_id="trig__{}__f_{}__l_{}".format(
                timezone.utcnow().isoformat(),
                value,
                value
            ),
            conf={
                "pipeline": kwargs['dag_run'].conf['analysis_pipeline'],
                "output": "{}/f-{}__l-{}".format(output, value, value),
                "input": kwargs['dag_run'].conf['input'],
                "data_file": data_file,
                "first": value,
                "last": value,
            }
        )
        triggers.append(trigger)


trigger_analysis_task = PythonOperator(
    dag=dag,
    task_id='trigger_analysis',
    provide_context=True,
    python_callable=trigger_analysis
)

trigger_analysis_task.set_upstream(illum)
```

And now our *cellprofiler-analysis-dag.py*.

```
# dags/cellprofiler-analysis-dag.py
from airflow import DAG
from datetime import datetime, timedelta
from airflow.operators.python_operator import PythonOperator
from airflow.operators.docker_operator import DockerOperator
import os
from pprint import pprint

this_env = os.environ.copy()

this_dir = os.path.dirname(os.path.realpath(__file__))

default_args = {
    'owner': 'airflow',
    'depends_on_past': False,
    'start_date': datetime(2019, 1, 1),
    'email': ['airflow@example.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG('cellprofiler_analysis', default_args=default_args, schedule_interval=None)

analysis = DockerOperator(
    dag=dag,
```

(continues on next page)

(continued from previous page)

```
    task_id='analysis',
    retries=1,
    volumes=[
        # Volumes are from the HOST MACHINE
        # UPDATE THIS to your path!
        '/path-on-HOST/data:/data'
    ],
    tty=True,
    image='cellprofiler',
    command=[
        "bash", "-c",
        """cellprofiler --run --run-headless \
            -p {{ dag_run.conf['pipeline'] }}  \
            -o {{ dag_run.conf['output'] }}  \
            -i {{ dag_run.conf['input'] }} \
            --data-file {{ dag_run.conf['data_file'] }} \
            -c -r -f {{ dag_run.conf['first'] }} -l {{ dag_run.conf['last'] }}"""
    ]
)


def gather_results(ds, **kwargs):
    """
    Once we have the Cellprofiler results let's do something with them!
    :param ds:
    :param kwargs:
    :return:
    """
    pass


gather_results_task = PythonOperator(
    dag=dag,
    task_id='gather_results_task',
    provide_context=True,
    python_callable=gather_results
)

gather_results_task.set_upstream(analysis)
```

Make sure that you update the *DockerOperator volumes* to match your local filesystem! Otherwise your analysis will not work!

9.7 Analysis Organizational Overview

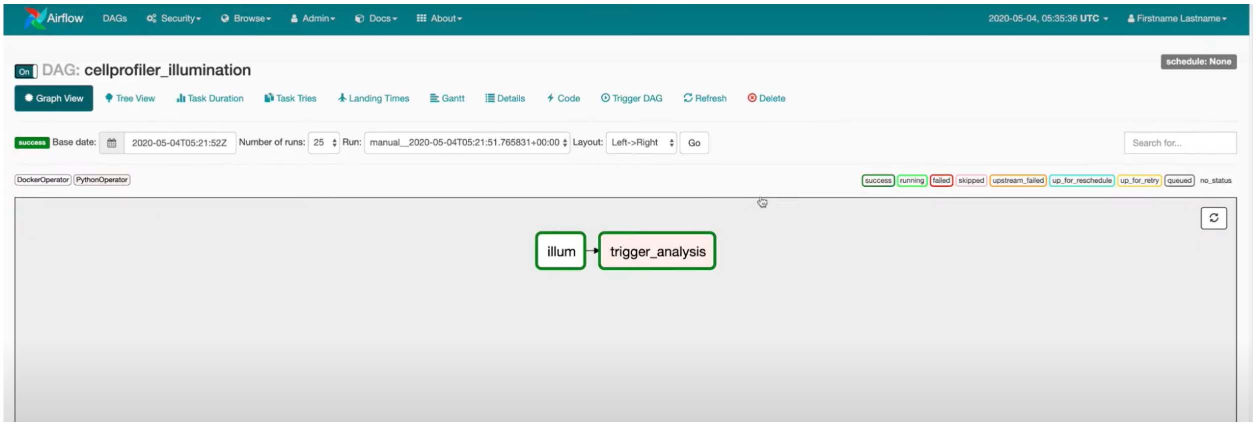
What we have here are 2 separate DAGs, one for each CellProfiler Analysis.

The steps are:

- Process Illumination pipeline for ALL images
- Grab the datafile to see how many images we have
- Dynamically generate one CellProfiler Airflow submission per image.
- (Placeholder) Do something with our results!

You will notice that we are dynamically splitting the CellProfiler analysis. This will get our analysis done faster, which is increasingly important when you buy fancy robotic microscopes that generate oodles of data. Airflow takes care of the job queueing under the hood, so all we need to do is to figure out the logic of how we split the analysis. If you'd like to know more about how parallelism is handled in Airflow this is a great article.

Ok, that last one is just a place holder. I would imagine that you would want to do something with your results once you have them, such as put them in a database, fire off one or more analyses based on certain criteria, or do some post processing, but for now this is blank so you can wonder about the possibilities.



9.8 Passing arguments to our Analysis

This was not obvious to me when I started using Airflow, and now you get to hear all about it! ;-)

You pass arguments to a Airflow using the conf object or argument. Depending on the operator type this might look slightly different. If you are using the Python operator you access it as a dict, *kwargs['dag_run'].conf*, and if you are using *Bash*, or in this case *Docker* you access it as a templated variable ` {{ dag_run.conf['variable'] }}`.

I will take you through how you use the Airflow web interface to trigger your DAG and pass in variables, but you can also trigger your DAGs using a REST API, through the Airflow CLI, or programatically using Python code. No matter how you trigger your DAG, you pass the configuration variables in as a JSON string.

9.9 A note on Docker Volumes

This is a little weird because we are using docker-compose to run Airflow, and then using the Docker Operator. Just keep in mind that when you use the Docker Operator you map your volumes using the paths on your HOST machine, not in the docker-compose containers. Your host machine is the machine you ran *docker-compose up* on.

For example:

```
analysis = DockerOperator(
    ...
    volumes=[
        # Volumes are from the HOST MACHINE
        '/path-on-HOST/data:/data'
        # NOT
        # '/data:/data'
        # Even though in our docker-compose instance the volume is bound as /data
    ],
```

(continues on next page)

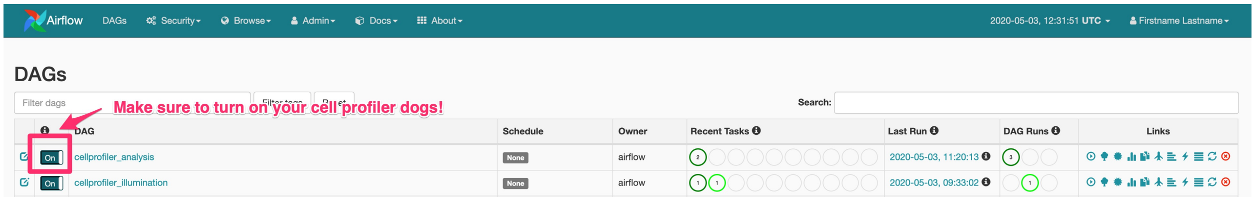
(continued from previous page)

```
    ...
)
```

9.10 Analyze!

We're ready! Now that you have your dags (you need to have your dags) all setup we can start to analyze data!

Login at *localhost:8080* with user *user* and password *bitnami*. You should see a screen that looks like this:



DAGs

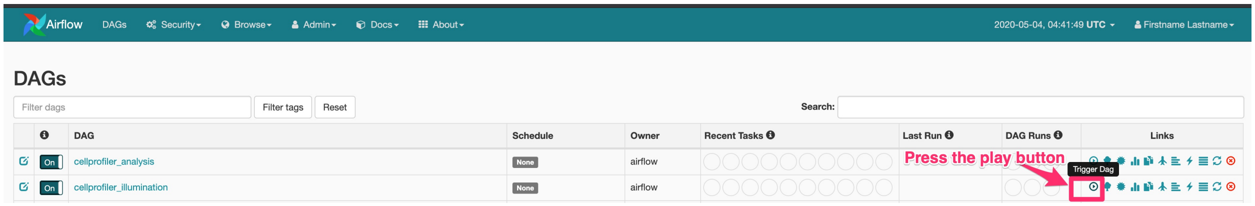
Make sure to turn on your cell profiler dogs!

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	On	cellprofiler_analysis	None	airflow	2	2020-05-03, 11:20:13	3	
	On	cellprofiler_illumination	None	airflow	1 1	2020-05-03, 09:33:02	1	

By default your dags will be off. Make sure to turn them on before proceeding to the next step!

9.11 Trigger your Analysis

Head to your main page at *localhost:8080* and trigger the *cellprofiler-illum* DAG.



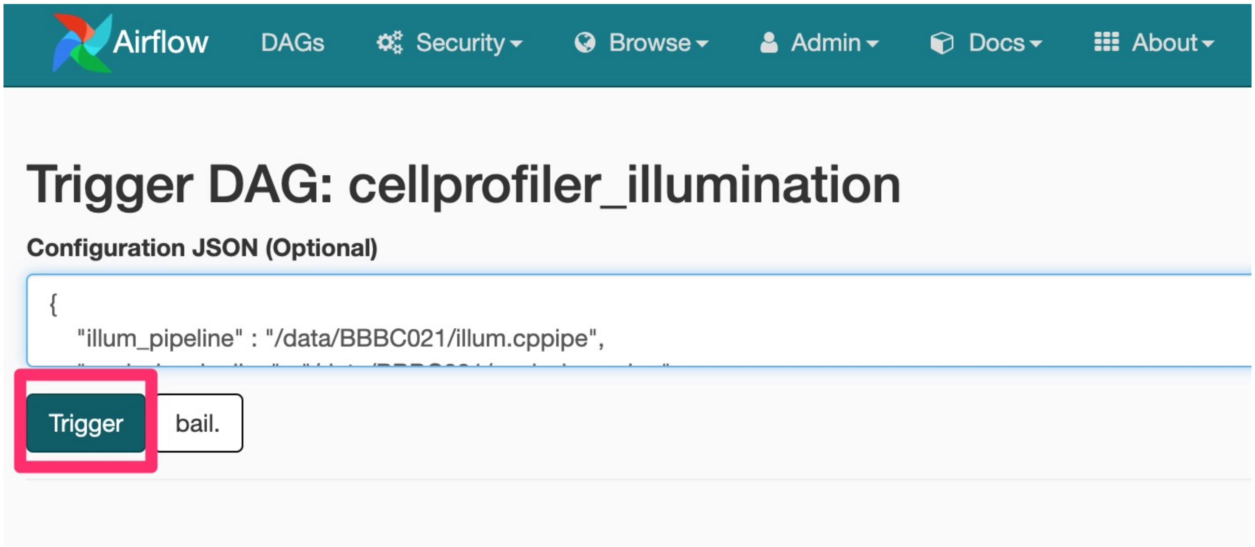
DAGs

Press the play button

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	On	cellprofiler_analysis	None	airflow				
	On	cellprofiler_illumination	None	airflow				

You will be prompted to add in some JSON configuration variables.

```
{
    "illum_pipeline" : "/data/BBBC021/illum.cppipe",
    "analysis_pipeline" : "/data/BBBC021/analysis.cppipe",
    "pipeline" : "/data/BBBC021/illum.cppipe",
    "output": "/data/BBBC021/Week1/Week1_22123",
    "input": "/data/BBBC021/Week1/Week1_22123",
    "data_file": "/data/BBBC021/images_week1.csv"
}
```

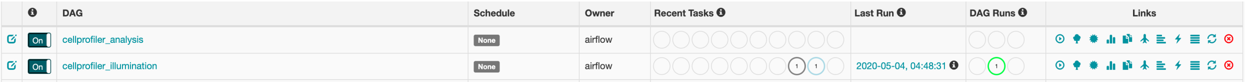


Trigger DAG: cellprofiler_illumination

Configuration JSON (Optional)

```
{
    "illum_pipeline" : "/data/BBBC021/illum.cppipe",
```

You'll be brought back to the main page and should see that your CellProfiler illumination analysis is running!



	ℹ	DAG	Schedule	Owner	Recent Tasks ℹ	Last Run ℹ	DAG Runs ℹ	Links
	On	cellprofiler_analysis	None	airflow				
	On	cellprofiler_illumination	None	airflow	1 1	2020-05-04, 04:48:31 ℹ	1	

9.12 Next Steps

I don't demonstrate any next steps here, because this book is already getting longer than I anticipated, but here are some ideas for you to follow up with.

- Push the analysis CSVs to S3
- Use Pandas to read in the CSVs and then store to a database with the Pandas Dataframe to_sql function and make sure to pay attention to whether or not you are creating the table, appending records, or truncating records.
- Run some kind of aggregate analysis on your dataset.
- Create a DAG to plot distributions of various datapoints.

9.13 Wrap Up

That's it! Let Airflow run and you will have your Illumination and Analysis pipelines running, all split nicely and something that isn't you babysitting keeping track of the job queue, logging, and success/failure rate. You can also integrate Airflow with any other system, such as a LIMS, reporting database, or secondary analysis workflow by using the REST API, the CLI, or code.

I would also like to give special thanks to the CellProfiler team, who gave me great input into this article and allowed me to guest post on their website.

9.14 Citations

https://data.broadinstitute.org/bbbc/BBBC021/

"We used image set BBBC021v1 [Caie et al., Molecular Cancer Therapeutics, 2010], available from the Broad Bioimage Benchmark Collection [Ljosa et al., Nature Methods, 2012]."

CHAPTER

TEN

CASE STUDY - SINGLE CELL DATA VISUALIZATION WITH CELLXGENE

The world of data visualization in Bioinformatics gets more and more interesting as time goes on. In this case study we'll discuss how I would use AWS to develop and deploy a dataset with the [CellxGene].

I want to make a quick side note here about data visualization in general. In this particular example we are going over [CellxGene], but these same principles apply to any data visualization application. [RShiny], for R, and [Dash], for Python, are both excellent platforms to interact with your datasets in real time. This idea of real time data analytics has taken off in the data analytics space, and with that comes the need to scale. The most interesting aspect of these frameworks is that you can deploy a truly hybrid, high availability data visualization platform that can analyze large datasets, on local storage, cloud storage (S3, gcp buckets) in real-time in a browser.

10.1 AWS Infrastructure

Let's take an example. Let's say you have a dataset, and you need to create a dashboard with several analysis types, with several options for parameters of the analysis. This application could consist of several layers.

10.1.1 Data Layer

You have your data stored on a networked AWS [EFS] file system, a [S3] bucket, or some other manner of remote storage.

10.1.2 Compute Layer

We need one or more [EC2] instances, hopefully delivered to us in some autoscaling fashion. I tend to separate out my development vs production environments, as I like something I can be very hands on with during my development, whereas for production I want a web application that *no one can mess with*.

10.1.3 Development

You have a [JupyterhubOnKubernetes] or [AWSParallelCluster] with [Jupyterhub] installed with enough memory and CPU. Optionally, if you wanted to keep it simple you could use a single EC2 instance or even your own laptop if your dataset allows for it. Our Jupyterhub should come along with all the tools we need for a [Cellxgene] data portal, including [AnnData] and [Scanpy]. I am going to use an image from my BioHub Collection, the Scanpy Notebook image. In development I'll always assume that I'm going to be doing at least some level of analysis, and that I need a lot of freedom to start/stop/restart instances.

10.1.4 Production

For production I like to use [Kubernetes] on AWS with [EKS]. It's not necessary for this dataset, but if necessary I will setup an Autoscaling Dask Cluster in order to scale my data visualizations. I'll often use the same Docker image as I develop with, in this case the Scanpy Notebook image, mostly because I really just can't be bothered to build 2 different docker images.

10.2 Application Layer

Your CellxGene application is a separate entity, but can also be configured for high availability, multiple instances, multiple users, to read data locally or through a remote protocol, etc. If you would like to have an indepth tutorial on autoscaling any application with Kubernetes check out my Autoscaling Dask Cluster tutorial.

10.3 Deployment and Site Reliability

Deployment and site reliability starts with Docker and the AWS Kubernetes (EKS) and ends with the sky as the limit. You can aggregate your logs, set up usage alerts, and monitor your memory and CPU usage. I like [Kubernetes] with [EKS] deployed with [Rancher] because it is an extremely well-supported way to deploy just about anything, which provides a uniform mechanism for deployment of your compute layer and applications.

10.4 A quick word about options

As we are all familiar with, in the computing space there are about a million ways to do any one thing. Of course, the options I describe here are not your only options. They are the ones I personally prefer and use on a daily basis.

10.5 Analyse your Dataset

To start off with you will need some way to access the Scanpy software. I'll be using the data available from the excellent Scanpy Tutorial.

If you're on an HPC you can run the following to bring up a docker container with your data directory. If you're on an [JupyterhubOnKubernetes] you'll need to ask your admin to add an image with Scanpy and choose it from your launch menu.

```
docker run -it -p 8888:8888 -v "$(pwd)/data:/home/jovyan/data" \
    dabbleofdevops/scanpy-notebook bash -c "jupyter notebook --port 8888 --ip 0.0.0.0"
```

10.5.1 Interact with your Dataset on Jupyterhub

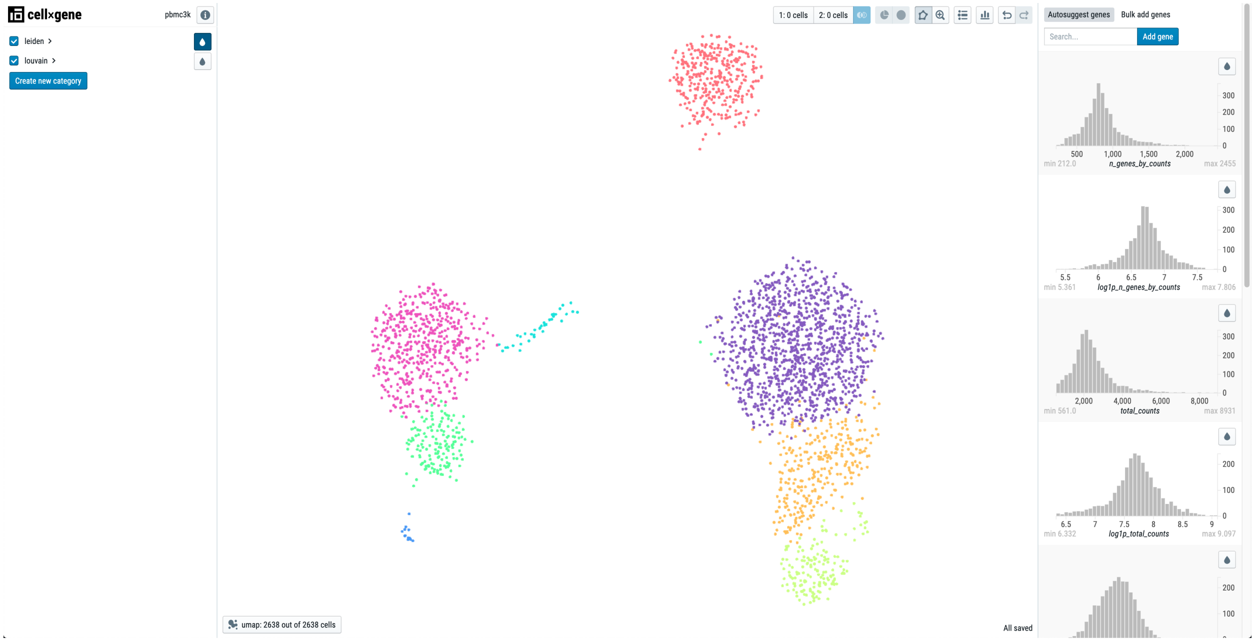
Once we somehow have access to the software and data we can start to interact with the data, create plots, and generally make sure everything is ok.

10.5.2 Spin up the Cellxgene Application

We can bring up [Cellxgene] directly from the commandline and proxy it within Jupyterhub. This is especially cool on an HPC instance because no more crazy SSH tunnels! In this example I am showing how you would bring up a remote dataset, but you can also point it towards a local dataset.

```
cellxgene launch --host 0.0.0.0 https://cellxgene-example-data.czi.technology/pbmc3k.
↪h5ad
```

Open up your browser to *${host}/user/${USER}/proxy/5005/* and you will see your application up.



If you'd like to learn more about proxying custom web services in Jupyterhub you can check out my HelpDesk.

Because we're still in the analysis development stage let's just say you go through several rounds of querying, plotting, saving, and restarting cellxgene. Finally, you get to a dataset you are happy with. You want to deploy this in production as a standalone web application that anyone in the world, or maybe just your organization can access.

10.6 Deploy Cellxgene on Kubernetes

Like I've stated so much that you're probably sick of hearing it by now, I like to deploy my production applications on Kubernetes. I get all the scaling, monitoring, and fault tolerance I could ever ask for.

I do not want to get too much into the techy details on deploying an application with Kubernetes, but I do have a blog post that covers it lots of detail. Instead, I am going to give a high level overview of what is needed in order to deploy an application on Kubernetes.

First, you'll need a [Kubernetes] cluster. Since you're reading this book I'll assume that it's on AWS, preferably deployed with [Rancher]. Then, you'll need a [Helm] Chart. Helm is the package manager for Kubernetes. The scientific community's contributing Helm charts is actually what made me move on over to Kubernetes in the first place. Initially, I kind of just rolled my eyes at yet another cluster, and kept on keeping on with the Kubernetes predecessor, Docker Swarm.

There are a few critical points you need to know when creating, or more likely, modifying a Helm Chart. First, you need a docker image that your application will run on. In this case this will be the Scanpy image. Every docker image also has a tag associated to it. For this example we'll just use the latest tag.

We also need to know the command to start the application, along with the port that the application is running on.

```
image:
  repository: dabbleofdevops/scanpy-notebook
  tag: latest
  pullPolicy: IfNotPresent
command: "cellxgene launch --host 0.0.0.0 https://cellxgene-example-data.czi.
↪technology/pbmc3k.h5ad"
containerPort: 5000
```

That's it. That is the information that we need to know to deploy our application to Kubernetes and package it into a Helm chart. When I am starting out I will usually just use the Bitnami NGINX Helm chart. One of the excellent things about Helm is that all of the configuration parameters are passed to the command line, making for a very handy installation.

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm update --install cellxgene-myanlysis bitnami/nginx \
    --set image.repository=dabbleofdevops/scanpy-notebook \
    --set image.tag=latest
    --set command="cellxgene launch --host 0.0.0.0 https://cellxgene-example-data.czi.
↪technology/pbmc3k.h5ad"
    --set containerPorts.http=5000
    --values
```

Once that's up you can query your Kubernetes cluster to get the service URLs, and even map those back to a proper domain name. You can setup autoscaling, environmental variables, init containers, volume mounts from [EFS], init containers to sync from S3, etc.

CHAPTER

ELEVEN

REFERENCES

Each citation includes the tagline or description from each of the projects, because I feel that it is important to represent these projects in their own words.

11.1 Bioinformatics Projects

[HumanGenomeProject] Human Genome Project (HGP) was one of the great feats of exploration in history. Rather than an outward exploration of the planet or the cosmos, the HGP was an inward voyage of discovery led by an international team of researchers looking to sequence and map all of the genes – together known as the genome – of members of our species, Homo sapiens. Beginning on October 1, 1990 and completed in April 2003, the HGP gave us the ability, for the first time, to read nature's complete genetic blueprint for building a human being.

[gnomAD] gnomAD - The Genome Aggregation Database (gnomAD) is a resource developed by an international coalition of investigators, with the goal of aggregating and harmonizing both exome and genome sequencing data from a wide variety of large-scale sequencing projects, and making summary data available for the wider scientific community.

[HumanCellAtlas] The Human Cell Atlas mission is to create comprehensive reference maps of all human cells—the fundamental units of life—as a basis for both understanding human health and diagnosing, monitoring, and treating disease.

[HumanCellAtlasDataPortal] Human Cell Atlas Data Portal supplies community generated, multi-omic, open data processed by standardized pipelines

11.2 Parallel Computing Libraries

[Dask] Dask - Scalable Analytics in Python

[ApacheSpark] Apache Spark Unified Analytics Engine for Big Data

11.3 Data Visualization Libraries

[Dash] Dash empowers teams to build data science and ML apps that put the power of Python, R, and Julia in the hands of business users.

[RShiny] RShiny is an R package that makes it easy to build interactive web apps straight from R.

[RStudioServer] RStudio Server enables you to provide a browser based interface to a version of R running on a remote Linux server, bringing the power and productivity of the RStudio IDE to server-based deployments of R. [Matplotlib] Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

[Seaborn] Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

[Bokeh] Bokeh is a Python library for creating interactive visualizations for modern web browsers

11.4 Scientific Computing Formats and File Types

[Xarray] Xarray - N-D labeled arrays and datasets in Python

[Zarr] Zarr is a format for the storage of chunked, compressed, N-dimensional arrays

[ApacheParquet] Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

[NetCDF] NetCDF is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.

[HDF5] HDF5 is a high-performance data management and storage suite

11.5 Infrastructure as Code

[Terraform] Terraform use Infrastructure as Code to provision and manage any cloud, infrastructure, or service

[CloudFormation] AWS CloudFormation gives you an easy way to model a collection of related AWS and third-party resources, provision them quickly and consistently, and manage them throughout their lifecycles, by treating infrastructure as code.

11.6 Testing Frameworks

[PyTest] The PyTest framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

[Bats] Bats is a TAP-compliant testing framework for Bash. It provides a simple way to verify that the UNIX programs you write behave as expected.

11.7 Documentation Engines

[Sphinx] Sphinx is a documentation generator or a tool that translates a set of plain text source files into various output formats, automatically producing cross-references, indices, etc.

11.8 AWS Services

[Lightsail] Lightsail

[EKS] Amazon Elastic Kubernetes Service (Amazon EKS), is a fully managed Kubernetes service.

[AWSParallelCluster] AWS ParallelCluster is an AWS-supported open source cluster management tool that makes it easy for you to deploy and manage High Performance Computing (HPC) clusters on AWS.

[AWSBatch] AWS Batch cluster enables developers, scientists, and engineers to easily and efficiently run hundreds of thousands of batch computing jobs on AWS.

[Fargate] AWS Fargate is a serverless compute engine for containers that works with both Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).

[Beanstalk] AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.

[Sagemaker] AWS SageMaker is a fully managed service that provides every developer and data scientist with the ability to build, train, and deploy machine learning (ML) models quickly.

[AutoScaling] AWS Autoscaling Groups

[S3] AWS S3

[Glacier] AWS S3 Glacier

[EFS] AWS EFS

[EC2] AWS EC2

11.9 Data Science Projects Using S3

[S3FS] Dask S3FS builds on botocore to provide a convenient Python filesystem interface for S3.

[S3Fuse] S3Fuse is a FUSE-based file system backed by Amazon S3.

[DistributedCellProfiler] Distributed Cellprofiler runs encapsulated docker containers with CellProfiler in the Amazon Web Services infrastructure.

[AnnData] AnnData is a project that provides a scalable way of keeping track of data and learned annotations, using an [Xarray]__ backend that supports [HDF5]__ and [Zarr]__ files.

11.10 Machine Learning and Artificial Intelligence Frameworks

[Tensorflow] Tensorflow is an end-to-end open source machine learning platform.

[PyTorch] PyTorch is an open source machine learning framework that accelerates the path from research prototyping to production deployment.

[H20] H20 AI is the open source leader in AI and machine learning with a mission to democratize AI for everyone.

11.11 Others

[Kubernetes] Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

[Rancher] Rancher From datacenter to cloud to edge, Rancher lets you deliver Kubernetes-as-a-Service.

[Helm] Helm is the package manager for Kubernetes.

[Minikube] Minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes.

[Jupyterhub] Jupyterhub brings the power of notebooks to groups of users. It gives users access to computational environments and resources without burdening the users with installation and maintenance tasks.

[JupyterhubOnKubernetes] The Jupyterhub On Kubernetes project will help you set up your own Jupyterhub on a cloud/on-prem k8s environment and leverage its scalable nature to support a large group of users.

[RStudio] RStudio Server enables you to provide a browser based interface to a version of R running on a remote Linux server, bringing the power and productivity of the RStudio IDE to server-based deployments of R.

[Bitnami] Bitnami makes it easy to get your favorite open source software up and running on any platform, including your laptop, Kubernetes and all the major clouds.

[SLURM] SLURM HPC Scheduler

[LabelStudio] LabelStudio is an Open Source Data Labeling Tool.

[GCS] Google Cloud

11.12 Workflow Managers

[ApacheAirflow] Apache Airflow is a platform created by the community to programmatically author, schedule and monitor workflows.

[Snakemake] With Snakemake , data analysis workflows are defined via an easy to read, adaptable, yet powerful specification language on top of Python.

[Cromwell] Cromwell is a Workflow Management System geared towards scientific workflows.

[Nextflow] Nextflow enables scalable and reproducible scientific workflows using software containers.

11.13 Bioinformatics Software and File Formats

[Samtools] Samtools is a suite of programs for interacting with high-throughput sequencing data.

[GATK] GATK is Developed in the Data Sciences Platform at the Broad Institute, the toolkit offers a wide variety of tools with a primary focus on variant discovery and genotyping. Its powerful processing engine and high-performance computing features make it capable of taking on projects of any size.

[VCF] VCF Format - VCF is a text file format (most likely stored in a compressed manner). It contains meta-information lines, a header line, and then data lines each containing information about a position in the genome. The format also has the ability to contain genotype information on samples for each position.

[Hail] Hail - Powering genomic analysis, at every scale. An open-source library for scalable genomic data exploration.

[Seurat] Seurat - R toolkit for single cell genomics.

[Plink] Plink - PLINK is a free, open-source whole genome association analysis toolset, designed to perform a range of basic, large-scale analyses in a computationally efficient manner.

[Scanpy] Scanpy Scanpy is a scalable toolkit for analyzing single-cell gene expression data built jointly with anndata. It includes preprocessing, visualization, clustering, trajectory inference and differential expression testing.

[Anndata] Anndata provides a scalable way of keeping track of data and learned annotations.

[CellProfiler] CellProfiler is free open-source software for measuring and analyzing cell images.

[Napari] Napari is a fast, interactive, multi-dimensional image viewer for Python. It's designed for browsing, annotating, and analyzing large multi-dimensional images.

[BWA] BWA is a software package for mapping low-divergent sequences against a large reference genome, such as the human genome.

[Loom] Loom is an efficient file format for large omics datasets.

[CellxGene] CellxGene is an interactive explorer for single-cell transcriptomics data.

11.14 Data Science Tools

[Conda] Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies.

[BioConda] Bioconda is a channel for the conda package manager specializing in bioinformatics software.

[Tidyverse] Tidyverse The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

[Pandas] Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

[Numpy] Numpy is the fundamental package for scientific computing with Python.

[ScikitLearn] Scikit Learn is a library for Machine Learning in Python.

11.15 Data Science Communities and Organizations

[PyData] PyData is a community for developers and users of open source data tools.

[NumFocus] NumFocus is a community working towards better tools to build a better world. From Netflix to NASA, researchers use our open source tools to solve the most challenging problems..

[Pangeo] Pangeo is a community platform for Big Data geoscience.

11.16 Consulting Companies

[Quansight] Quansight is building a data driven economy.

[Coiled] Coiled deploys Dask for everyone, everwhere.

[BioinformaticsCRO] Bioinformatics CRO offers Bioinformatics Done Right, Now.

[DabbleofDevOps] Dabble of DevOps is my consulting company. Come talk to me!

11.17 Disclaimer

I have made every effort to accurately cite each source, as well as describe that source using their own words. If you feel that I have either failed to source or sourced a project incorrectly, please get in touch with me at jillian@dabbleofdevops.com and I will fix it.

CHAPTER TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[HumanGenomeProject] Human Genome Project (HGP) was one of the great feats of exploration in history. Rather than an outward exploration of the planet or the cosmos, the HGP was an inward voyage of discovery led by an international team of researchers looking to sequence and map all of the genes – together known as the genome – of members of our species, Homo sapiens. Beginning on October 1, 1990 and completed in April 2003, the HGP gave us the ability, for the first time, to read nature's complete genetic blueprint for building a human being.

[gnomAD] gnomAD - The Genome Aggregation Database (gnomAD) is a resource developed by an international coalition of investigators, with the goal of aggregating and harmonizing both exome and genome sequencing data from a wide variety of large-scale sequencing projects, and making summary data available for the wider scientific community.

[HumanCellAtlas] The Human Cell Atlas mission is to create comprehensive reference maps of all human cells—the fundamental units of life—as a basis for both understanding human health and diagnosing, monitoring, and treating disease.

[HumanCellAtlasDataPortal] Human Cell Atlas Data Portal supplies community generated, multi-omic, open data processed by standardized pipelines

[Dask] Dask - Scalable Analytics in Python

[ApacheSpark] Apache Spark Unified Analytics Engine for Big Data

[Dash] Dash empowers teams to build data science and ML apps that put the power of Python, R, and Julia in the hands of business users.

[RShiny] RShiny is an R package that makes it easy to build interactive web apps straight from R.

[RStudioServer] RStudio Server enables you to provide a browser based interface to a version of R running on a remote Linux server, bringing the power and productivity of the RStudio IDE to server-based deployments of R.

[Matplotlib] Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

[Seaborn] Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

[Bokeh] Bokeh is a Python library for creating interactive visualizations for modern web browsers

[Xarray] Xarray - N-D labeled arrays and datasets in Python

[Zarr] Zarr is a format for the storage of chunked, compressed, N-dimensional arrays

[ApacheParquet] Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language.

[NetCDF] NetCDF is a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data.

[HDF5] HDF5 is a high-performance data management and storage suite

[Terraform] Terraform use Infrastructure as Code to provision and manage any cloud, infrastructure, or service

[CloudFormation] AWS CloudFormation gives you an easy way to model a collection of related AWS and third-party resources, provision them quickly and consistently, and manage them throughout their lifecycles, by treating infrastructure as code.

[PyTest] The PyTest framework makes it easy to write small tests, yet scales to support complex functional testing for applications and libraries.

[Bats] Bats is a TAP-compliant testing framework for Bash. It provides a simple way to verify that the UNIX programs you write behave as expected.

[Sphinx] Sphinx is a documentation generator or a tool that translates a set of plain text source files into various output formats, automatically producing cross-references, indices, etc.

[Lightsail] Lightsail

[EKS] Amazon Elastic Kubernetes Service (Amazon EKS), is a fully managed Kubernetes service.

[AWSParallelCluster] AWS ParallelCluster is an AWS-supported open source cluster management tool that makes it easy for you to deploy and manage High Performance Computing (HPC) clusters on AWS.

[AWSBatch] AWS Batch cluster enables developers, scientists, and engineers to easily and efficiently run hundreds of thousands of batch computing jobs on AWS.

[Fargate] AWS Fargate is a serverless compute engine for containers that works with both Amazon Elastic Container Service (ECS) and Amazon Elastic Kubernetes Service (EKS).

[Beanstalk] AWS Elastic Beanstalk is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS.

[Sagemaker] AWS SageMaker is a fully managed service that provides every developer and data scientist with the ability to build, train, and deploy machine learning (ML) models quickly.

[AutoScaling] AWS Autoscaling Groups

[S3] AWS S3

[Glacier] AWS S3 Glacier

[EFS] AWS EFS

[EC2] AWS EC2

[S3FS] Dask S3FS builds on botocore to provide a convenient Python filesystem interface for S3.

[S3Fuse] S3Fuse is a FUSE-based file system backed by Amazon S3.

[DistributedCellProfiler] Distributed Cellprofiler runs encapsulated docker containers with CellProfiler in the Amazon Web Services infrastructure.

[AnnData] AnnData is a project that provides a scalable way of keeping track of data and learned annotations, using an [Xarray] backend that supports [HDF5] and [Zarr] files.

[Tensorflow] Tensorflow is an end-to-end open source machine learning platform.

[PyTorch] PyTorch is an open source machine learning framework that accelerates the path from research prototyping to production deployment.

[H20] H20 AI is the open source leader in AI and machine learning with a mission to democratize AI for everyone.

[Kubernetes] Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.

[Rancher] Rancher From datacenter to cloud to edge, Rancher lets you deliver Kubernetes-as-a-Service.

[Helm] Helm is the package manager for Kubernetes.

[Minikube] Minikube is local Kubernetes, focusing on making it easy to learn and develop for Kubernetes.

[Jupyterhub] Jupyterhub brings the power of notebooks to groups of users. It gives users access to computational environments and resources without burdening the users with installation and maintenance tasks.

[JupyterhubOnKubernetes] The Jupyterhub On Kubernetes project will help you set up your own Jupyterhub on a cloud/on-prem k8s environment and leverage its scalable nature to support a large group of users.

[RStudio] RStudio Server enables you to provide a browser based interface to a version of R running on a remote Linux server, bringing the power and productivity of the RStudio IDE to server-based deployments of R.

[Bitnami] Bitnami makes it easy to get your favorite open source software up and running on any platform, including your laptop, Kubernetes and all the major clouds.

[SLURM] SLURM HPC Scheduler

[LabelStudio] LabelStudio is an Open Source Data Labeling Tool.

[GCS] Google Cloud

[ApacheAirflow] Apache Airflow is a platform created by the community to programmatically author, schedule and monitor workflows.

[Snakemake] With Snakemake , data analysis workflows are defined via an easy to read, adaptable, yet powerful specification language on top of Python.

[Cromwell] Cromwell is a Workflow Management System geared towards scientific workflows.

[Nextflow] Nextflow enables scalable and reproducible scientific workflows using software containers.

[Samtools] Samtools is a suite of programs for interacting with high-throughput sequencing data.

[GATK] GATK is Developed in the Data Sciences Platform at the Broad Institute, the toolkit offers a wide variety of tools with a primary focus on variant discovery and genotyping. Its powerful processing engine and high-performance computing features make it capable of taking on projects of any size.

[VCF] VCF Format - VCF is a text file format (most likely stored in a compressed manner). It contains meta-information lines, a header line, and then data lines each containing information about a position in the genome. The format also has the ability to contain genotype information on samples for each position.

[Hail] Hail - Powering genomic analysis, at every scale. An open-source library for scalable genomic data exploration.

[Seurat] Seurat - R toolkit for single cell genomics.

[Plink] Plink - PLINK is a free, open-source whole genome association analysis toolset, designed to perform a range of basic, large-scale analyses in a computationally efficient manner.

[Scanpy] Scanpy Scanpy is a scalable toolkit for analyzing single-cell gene expression data built jointly with anndata. It includes preprocessing, visualization, clustering, trajectory inference and differential expression testing.

[Anndata] Anndata provides a scalable way of keeping track of data and learned annotations.

[CellProfiler] CellProfiler is free open-source software for measuring and analyzing cell images.

[Napari] Napari is a fast, interactive, multi-dimensional image viewer for Python. It's designed for browsing, annotating, and analyzing large multi-dimensional images.

[BWA] BWA is a software package for mapping low-divergent sequences against a large reference genome, such as the human genome.

[Loom] Loom is an efficient file format for large omics datasets.

[CellxGene] CellxGene is an interactive explorer for single-cell transcriptomics data.

[Conda] Conda is an open source package management system and environment management system that runs on Windows, macOS and Linux. Conda quickly installs, runs and updates packages and their dependencies.

[BioConda] Bioconda is a channel for the conda package manager specializing in bioinformatics software.

[Tidyverse] Tidyverse The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

[Pandas] Pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

[Numpy] Numpy is the fundamental package for scientific computing with Python.

[ScikitLearn] Scikit Learn is a library for Machine Learning in Python.

[PyData] PyData is a community for developers and users of open source data tools.

[NumFocus] NumFocus is a community working towards better tools to build a better world. From Netflix to NASA, researchers use our open source tools to solve the most challenging problems..

[Pangeo] Pangeo is a community platform for Big Data geoscience.

[Quansight] Quansight is building a data driven economy.

[Coiled] Coiled deploys Dask for everyone, everwhere.

[BioinformaticsCRO] Bioinformatics CRO offers Bioinformatics Done Right, Now.

[DabbleofDevOps] Dabble of DevOps is my consulting company. Come talk to me!