Columbus Developer Manual

1. Introduction

Use of the PyData ecosystem for data cleaning and integration has become ubiquitous today. The PyData ecosystem has been so successful due to the following reasons:

1. Ease of installation: Services are packaged as PyPi packages that even a lay user can easily install.
2. Interoperability of services: It is very easy to call multiple python packages within an interactive Jupyter notebook environment and have them interoperate in order to accomplish particular tasks i.e. Services can be composed together to develop more complicated applications.
3. Extensibility: The PyData Ecosystem provides a very well-defined set of guidelines for developing new PyPi packages. If a particular application cannot be built with the available services, it is easy for developers to write and publish a new PyPi package for it, which can then be used by the community.

However, the PyData ecosystem suffers from some critical limitations, namely:

1. Not Collaborative: PyPi packages are typically designed to run within a local environment for a single user. In practice, data cleaning is often highly collaborative. Users are often forced to resorting to crude collaboration mechanisms such as mailing csv tables and python code to each other.
2. Not Scalable: Again, as PyPi packages are designed to run within a local environment within a single machine, they cannot scale beyond the memory, disk space and other resources provided by that environment.
3. Not Multilingual: Users are restricted to developing services and applications in python. Data scientists knowledgeable in other languages such as SQL or R cannot easily use these tools.
4. Not Seamless: Can only import packages written in python. If you happen to have a handy service written in some other programming environment, you would need to rewrite it from scratch as a PyPi package or write a python wrapper around it in order to use it in this ecosystem. This is particularly problematic because python programs are known to not be particularly performant.

Some cloud-based data cleaning and integration tools today offer solutions for scaling and collaboration. And cloud tools also provide rigid pipelines for composing services together. However, to the best of our knowledge, there does not exist a cloud-native solution which allows users to flexibly compose cloud-native scalable and collaborative services in a user-defined workflow. There also does not exist a cloud-native ecosystem equivalent to the PyData ecosystem in terms of the quantity and variety of available services and a well-defined guidelines for extensibility.

In order to address these concerns, we are building the Columbus project. The Columbus ecosystem is designed to be scalable, collaborative, multilingual and seamless. Columbus is designed to be entirely cloud-native. Services are expected to be containerized and deployed onto a Kubernetes cluster. Services expose a REST API and can interoperate through REST API calls on an internal network. Containerization and REST API together solves the seamless interoperability problem, as services can now be developed in any programming environment as long as they can be built into a container image that offers a published set of REST API calls for interoperability.

These were the goals we worked toward while developing the first version of Columbus and CDrive:

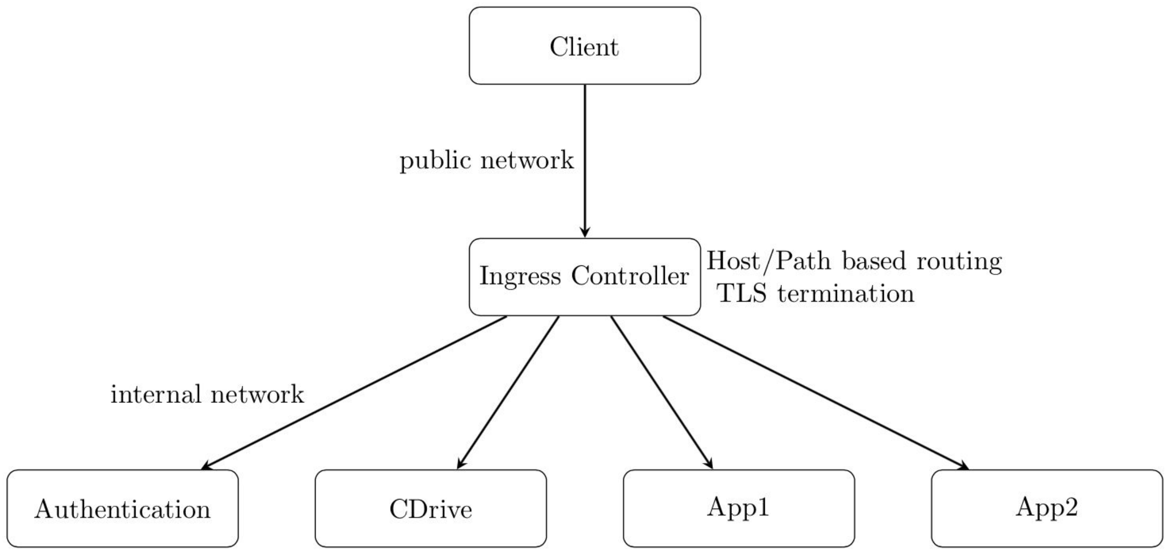
1. A data science team can set up Columbus on a cluster of machines.
2. Users can then create accounts, log in, upload and share data among each other through Columbus and the applications it offers.
3. Allow users to develop new applications that can operate on the data within user accounts. Common services that most applications require such as authentication, authorization, data sharing, TLS termination, load balancing, scaling etc. should be provided out of the box by Columbus to the app developers.
4. Users should be able to install these applications in their account and use them to operate on data within their account or data shared with them.

2. Architecture

2.1 Overview

As Columbus is designed to be deployed on Kubernetes, all internal Columbus services such as authentication, ingress controller, cdrive etc. as well as user applications are containerized and deployed in Kubernetes pods. Separate Kubernetes deployments and services are created for authentication, cdrive api, cdrive user interface and each individual application. Kubernetes connects all these services over an internal network. An ingress controller service exposes a public IP which acts as the single public entrypoint for clients to hit these services.

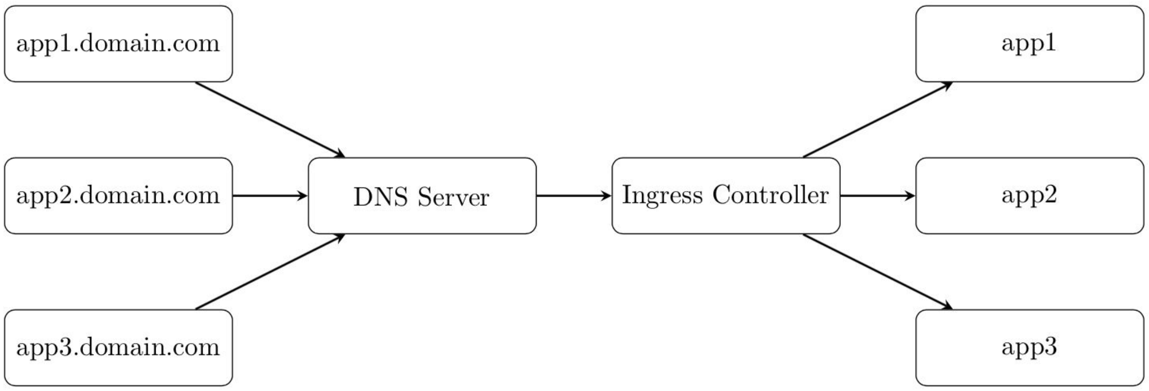
2.2 Routing



Columbus uses an Nginx Ingress Controller to route client requests to appropriate services. A Kubernetes ingress object is created which specifies the configuration rules that the ingress controller should use for routing requests.

2.2.1 Virtual Hosting

Virtual hosting is a method that allows multiple domain names to be served by a common pool of servers or in our case, by a single common Kubernetes cluster. Virtual hosting allows the server to share its resources such as memory and processor cycles without requiring all the services provided to use the same domain name. We have different domain/subdomain names for different services such as authentication (authentication.columbusecosystem.com), cdrive (cdrive.columbusecosystem.com), user applications (applications.columbusecosystem.com), and public applications (<app_name>.columbusecosystem.com). However, we add DNS A records pointing all these domain names to the public IP address of the Nginx Ingress Controller service. The ingress controller can now use the hostname to route the request to the appropriate service over the internal Kubernetes network.



The Nginx Ingress Controller uses a combination of hostname and path based routing. Hostname based routing is used to route requests to the authentication service, cdrive and public applications. All private applications are accessed under the same hostname. The username and application name are specified in the path. For example, when user_i installs app_j, Columbus launches an instance of app_j in a pod, creates a service for it and adds an entry to the Kubernetes Ingress resource which configures a proxy for this service object. Subsequently, any request to applications.columbusecosystem.com/user_i/app_j/… will be routed to user_i's instance of app_j by the ingress controller.

2.2.2 TLS termination

Alongside routing, the ingress controller also provides TLS termination, thus allowing all services to be served over HTTPS. The assumption here is that the Kubernetes internal network is inherently secure. Given this assumption, it is useful to have TLS termination implemented in a central location instead of each service having to implement it separately. This is beneficial to Columbus app developers as well, as their apps avail the additional security layer of HTTPS without any additional development effort.

Columbus uses Cert-Manager, Kubernetes' native certificate manager to automatically provision TLS certificates. We create a Cluster Issuer resource for issuing certificates. The Cert-Manager contacts the Cluster Issuer to request TLS certificates. The Cluster Issuer resource represents a certificate authority.

The Columbus Cluster Issuer is configured to use TLS certificates from Let's Encrypt, which is a free, automated and open certificate authority.

2.3 Authentication

Columbus uses the industry standard Oauth 2.0 protocol for authentication and authorization. Columbus has a separate authentication service which provides oauth endpoints for oauth workflows. Applications (including CDrive) can use this service for authentication and authorization. This process is identical to how websites and applications allow users to log in with their Google, Facebook or Github accounts.

2.4 Persistent Storage

Kubernetes, by default, runs containers as 'stateless' pods i.e. Kubernetes assumes that the containers do not have any persistent data within them. In such a scenario, whenever Kubernetes shuts down an application container and starts a new container, application data will be lost. Kubernetes is designed to do this occasionally for load balancing and several other reasons.

This problem is typically addressed on Kubernetes using a concept called 'Stateful Sets'. Applications configured as stateful sets, will attach a Kubernetes persistent volume to pods. These persistent volumes will last beyond the lifecycle of the pod, and any new pods for the application will connect to this persistent volume to recover state. Kubernetes persistent volumes are similar to docker volume mounts. But where Docker only allows mounting a volume to a host directory, Kubernetes offers various other alternatives such as EBS Volumes, NFS, Cephfs, Glusterfs, EFS etc.

In Columbus, we attach EBS Volumes to application pods as persistent volumes using a Persistent Volume Claim. This storage is dynamic in the sense that if a persistent volume is not bound to persistent volume claim, a new EBS volume is dynamically provisioned and bound to it. If a persistent volume is already bound to the claim, the stateful application attaches to it. So, Kubernetes can shut down and re-start these container many times but the data is always maintained as the same EBS volume is always re-attached.

3. Setup

We recommend running Columbus on AWS. Once you have an AWS account, you can proceed with provisioning your Kubernetes cluster and setting up Columbus on it.

3.1 Cluster Setup

We recommend using Kops, an official Kubernetes project for managing production grade Kubernetes clusters. Start out by installing command line tools for Kops, AWS, kubectl and helm.

Mac

```
brew update && brew install kops
brew install kubernetes-cli
brew install awscli
brew install kubernetes-helm
```

Linux

```
curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s aws
cov--https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_name |
cut -d '"' -f 4)/kops-linux-amd64
chmod +x ./kops
sudo mv ./kops /usr/local/bin/

curl -Lo kubectl https://storage.googleapis.com/kubernetes-release/release/$(curl
-s
https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd
64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl

pip install awscli

wget https://get.helm.sh/helm-v2.14.1-linux-amd64.tar.gz
tar -zxvf helm-v2.14.1-linux-amd64.tgz
mv linux-amd64/helm /usr/local/bin/helm
```

If you are using a windows machine, we recommend you launch a linux ec2 instance from the AWS console, ssh into it and use the linux commands above to launch your Kubernetes cluster.

Next, create an IAM user for kops. The user requires API credentials in order to use kops. Create the user and the credentials using the AWS console. The kops user will require the following IAM permissions to function properly: AmazonEC2FullAccess, AmazonRoute53FullAccess, AmazonS3FullAccess, IAMFullAccess, AmazonVPCFullAccess.
Record the AccessKeyID and SecretAccessKey that will be provided to you when you create the user and use them below:

```
aws configure
export AWS_ACCESS_KEY_ID=$(aws configure get aws_access_key_id)
export AWS_SECRET_ACCESS_KEY=$(aws configure get aws_secret_access_key)
```

Create an S3 bucket to store cluster state:

```
aws s3api create-bucket --bucket prefix-example-com-state-store --region us-east-1
```

Create a gossip-based cluster by making sure the cluster name ends with k8s.local, specifying the number of EC2 machines you want in the cluster and their type:

```
export NAME=myfirstcluster.k8s.local
export KOPS_STATE_STORE=s3://prefix-example-com-state-store
kops create cluster --zones us-west-2a --node-count=2 --node-size=m4.large ${NAME}
kops update cluster ${NAME} --yes
```

3.2 Columbus setup

Once your cluster is up and running, you can proceed with Columbus setup.

Log in to AWS and go to Route53 service. Register a domain. Let's call it columbusecosystem.com.

A kubernetes script, say script.yml can be applied with the following command:

```
kubectl apply -f script.yml
```

Create a cluster binding role by applying this script:
https://raw.githubusercontent.com/kaushikc92/columbus-kube-scripts/master/tiller-clusterrolebinding.yaml

Initialize Helm:

```
helm init --service-account tiller
```

Install cert-manager:

```
helm install stable/cert-manager \
--namespace kube-system
--set ingressShim.defaultIssuerName=letsencrypt-prod
--set ingressShim.defaultIssuerKind=ClusterIssuer
--version v0.5.2
```

Deploy the TLS certificate manager by entering your email and applying this script:
https://raw.githubusercontent.com/kaushikc92/columbus-kube-scripts/master/letsencrypt-staging.yaml

Install Nginx Ingress Controller:

```
helm install stable/nginx-ingress \
--namespace kube-system
--name nginx-ingress
```

Deploy the Nginx ingress controller by entering your domain name and applying this script:
https://raw.githubusercontent.com/kaushikc92/columbus-kube-scripts/master/my-ingress.yaml

Run the following command to find out the external IP of the ingress controller:

```
kubectl -n kube-system get svc ingress-nginx
```

Open Route53 on the AWS console. Navigate to the hosted zone for your domain name and DNS A records for authentication.<domain>, api.cdrive.<domain>, cdrive.<domain> and apps.<domain> pointing to the external IP of the ingress controller.

Setup authentication service by applying the following scripts:
https://raw.githubusercontent.com/kaushikc92/authentication/master/pvc.yml
https://raw.githubusercontent.com/kaushikc92/authentication/master/deployment.yml
https://raw.githubusercontent.com/kaushikc92/authentication/master/service.yml

Setup Cdrive service:
Go to authentication.<domain>/o/applications and register CDrive application. Note the client id and client secret.
Download and apply https://raw.githubusercontent.com/kaushikc92/cdrive/master/pvc.yml
Download https://raw.githubusercontent.com/kaushikc92/cdrive/master/deployment.yml , edit the environment variables values defined in the script and apply it
Download and apply https://raw.githubusercontent.com/kaushikc92/cdrive/master/service.yml
Download and apply https://raw.githubusercontent.com/kaushikc92/cdrive-ui/master/deployment.yml
Download and apply https://raw.githubusercontent.com/kaushikc92/cdrive-ui/master/deployment.yml

That's it! Now you can visit cdrive.<domain> and you should be able to access CDrive.

4. CDrive

4.1 Accounts

To create a new Columbus account, open CDrive home page (https://cdrive.columbusecosystem.com), click on the create account link and fill in your details. You can then login to CDrive.

4.2 Files

CDrive has an intuitive interface for uploading files from local machine. Once files have been uploaded, there are options for downloading, deleting or sharing the file in the actions column. Depending on the file type, there will also be options to open the file in certain Columbus applications.

4.3 Sharing

By clicking on the share option from a file's list of actions, a user can share the file with other users. The pop up dialog asks for the email address of the target user. A user can view all the files shared with him by clicking on the 'Shared' tab.

4.4 Applications

The applications tab contains a list of applications installed by the user. An initial set of applications will come pre-installed when a user creates a Columbus account. But the user can also install new applications from this tab. Clicking on install application opens up a list of applications from which the user can select and install into his CDrive. Once installed, the application will be available for the user to access from the applications tab.

5. Application Development

Applications will be served at apps.<domain>/user_i/app_j. So, the application needs to be developed in such a way that it can be served at this path. Note that multiple instances of the app need to be served for different values of ‘user_i’, so the app should be capable of being served at <domain>/*/app_j.

Applications need to first be registered by app developers in order to obtain the client id and client secret. In order to use the authentication service for his application, the app developer has to follow the following steps:

1. Register the application at authentication.columbusecosystem.com/o/applications. Specify authorization grant type as ‘authorization code’ and specify redirect url as <app_name>.columbusecosystem.com.
2. On registering the application, you will receive a client id and a client secret. Save it.
3. Create a ‘login with Columbus’ link or button on the application page. When a user clicks that link, direct the user to authentication.com/o/authorize/?response_type=code&client_id=<client_id>&redirect_uri=<redirect_url>&state=<random_string>
4. When the user clicks on login, he will see the Columbus login page. On logging in, the user will be prompted to authorize the application to access his CDrive data.
5. On authorizing, the user will be redirected to <redirect_url>/?code=<code>.
6. The application can now use this code parameter to obtain a token by making a post request to authentication.columbusecosystem.com/o/token with the following data: grant_type=authorization_code, code=<code>, redirect_uri=<redirect_uri>, client_id=<client_id>, client_secret=<client_secret>.
7. The authentication service will send the token, refresh token and token expiry time in response.
8. The application should store the token in session cookies. When the token expires, the application can use the refresh token to request a new token.