U.S. AIR FORCE

76TH SOFTWARE ENGINEERING GROUP

DEPARTMENT 70

Ignire Omnis

OKC EDDGE

Advanced Development | 76 Software Engineering

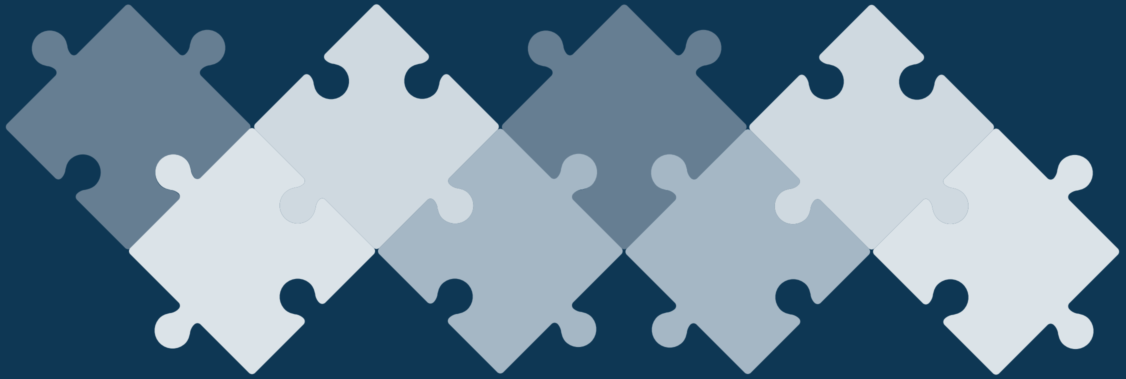
Reliable Modular Systems: An Implementation of the Dynamic-CFAR Algorithm

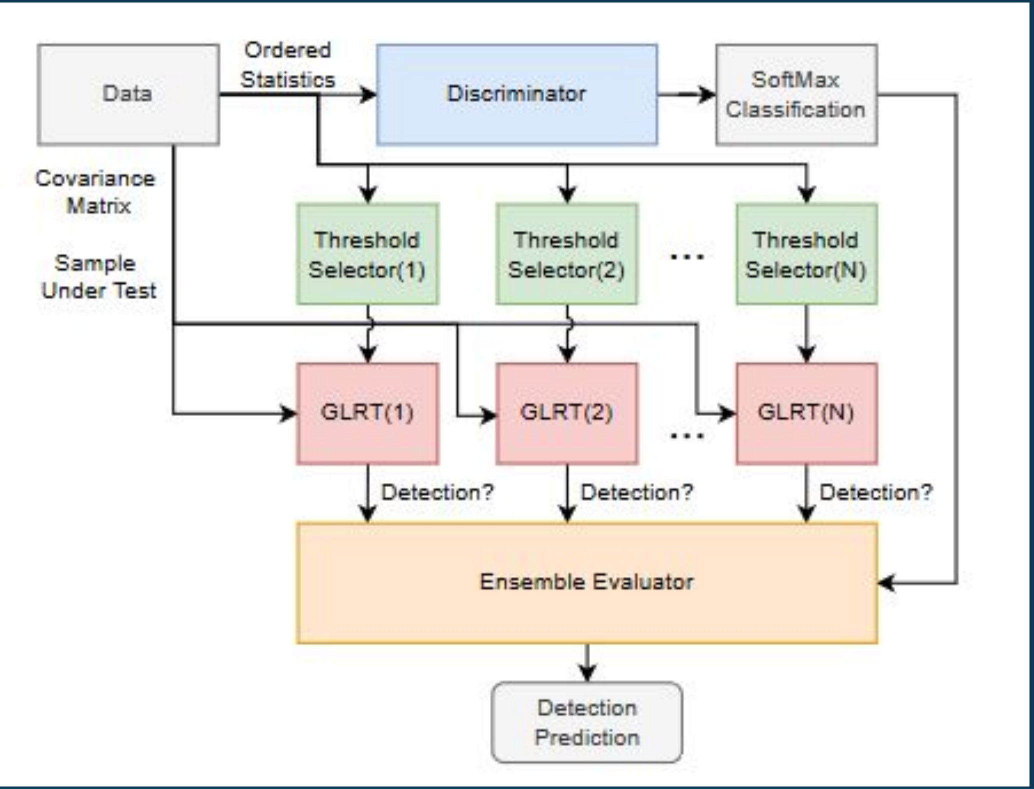
Ryan Fontaine, Lesya Borowska, Zak Kastl

Table of Contents:

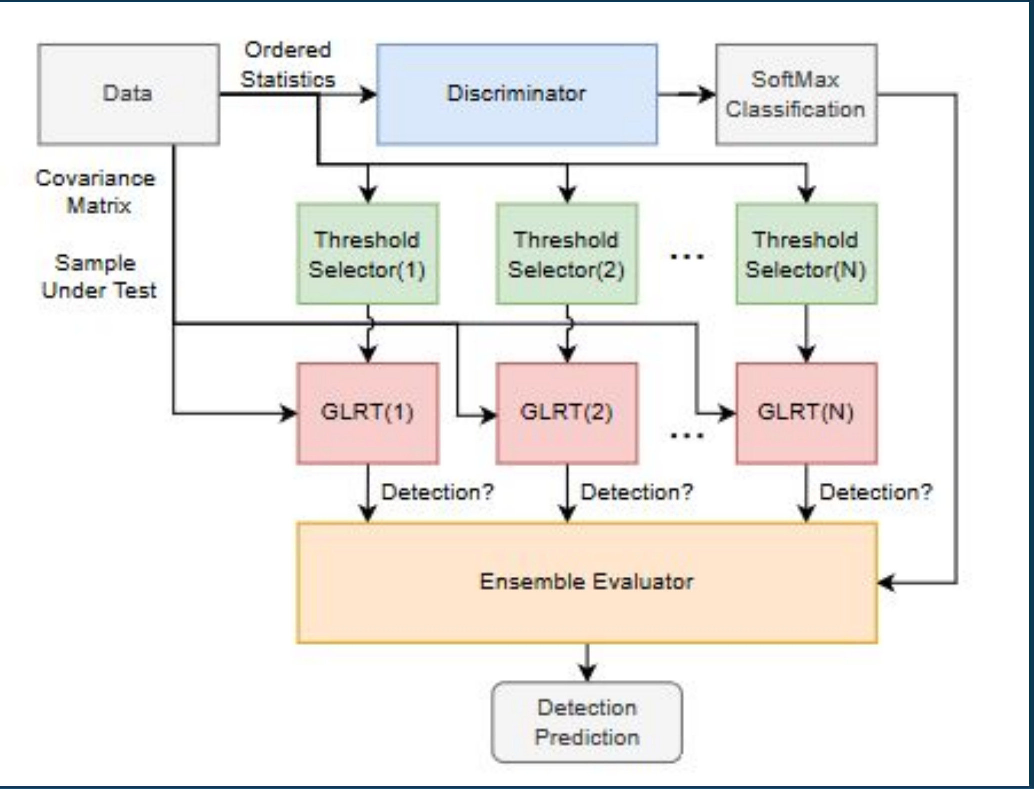
01 Objectives and Overview

A brief description of Docker and Kubernetes.





With input Radar Data,
The Dynamic-CFAR algorithm uses a Discriminator to classify and select an appropriate Prediction Model used to intelligently detect False Alarms.



With input Radar Data,
The Dynamic-CFAR algorithm uses a Discriminator to classify and select an appropriate Prediction Model used to intelligently detect False Alarms.

What if a component was not situationally applicable? This could lead to ineffectiveness or failure due to unknown data format.

Using a system of virtual ‘Containers’ a program can be run independent of operating system in a secure environment within the Docker architecture.

Images

A Docker Image is similar to an Operating System. For example, a Container may be running on an Ubuntu Image.

Containers

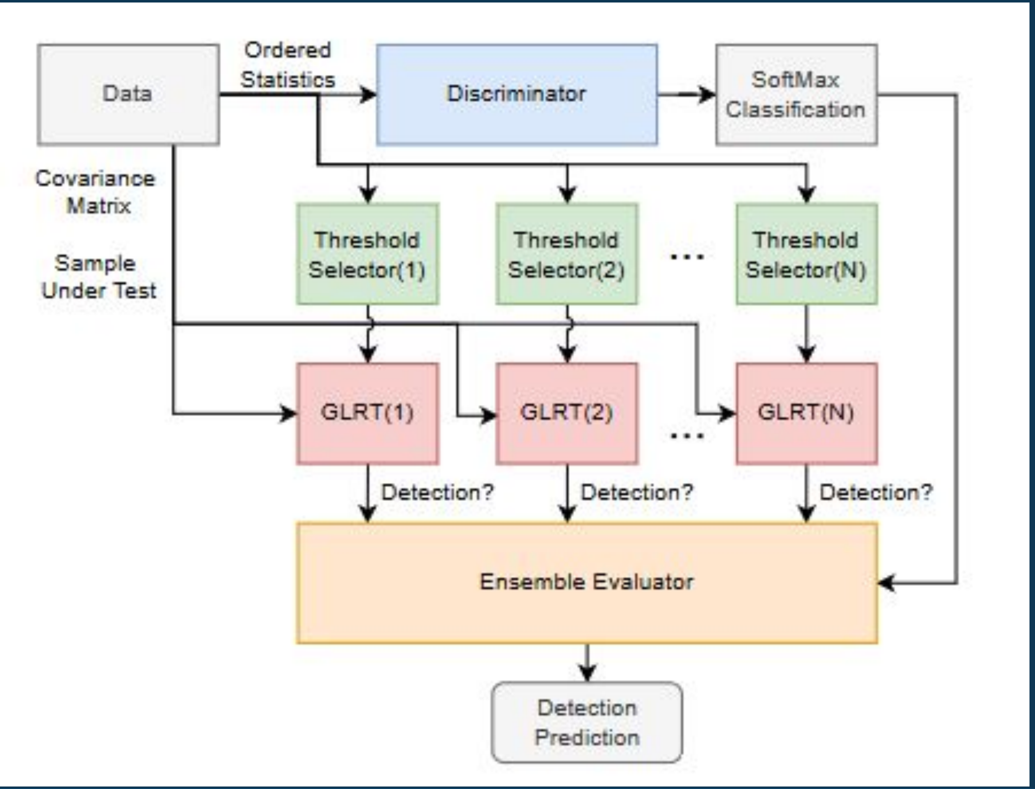
This is the main building block of Docker. It is basically a simplified Virtual Machine, which only runs one command. It is an instance of an image.

Volumes

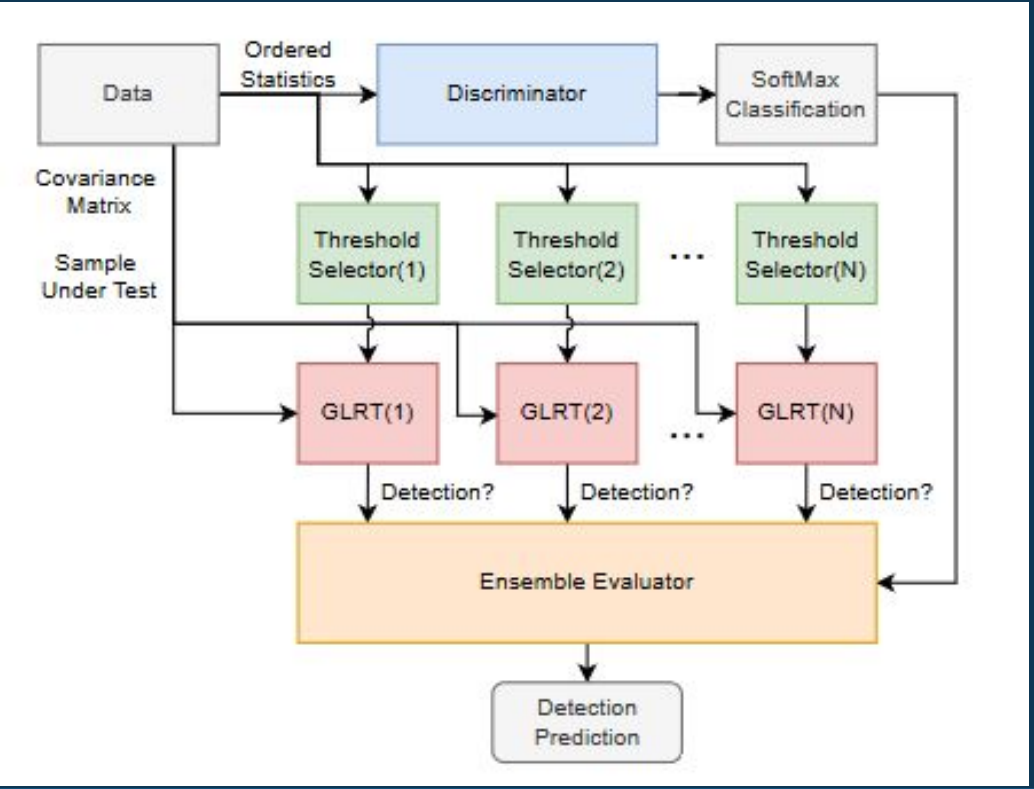
Containers are inherently isolated, as no created files are saved after running. Volumes can be attached to a container to create a shared file system that persists after runtime.

Building and Running

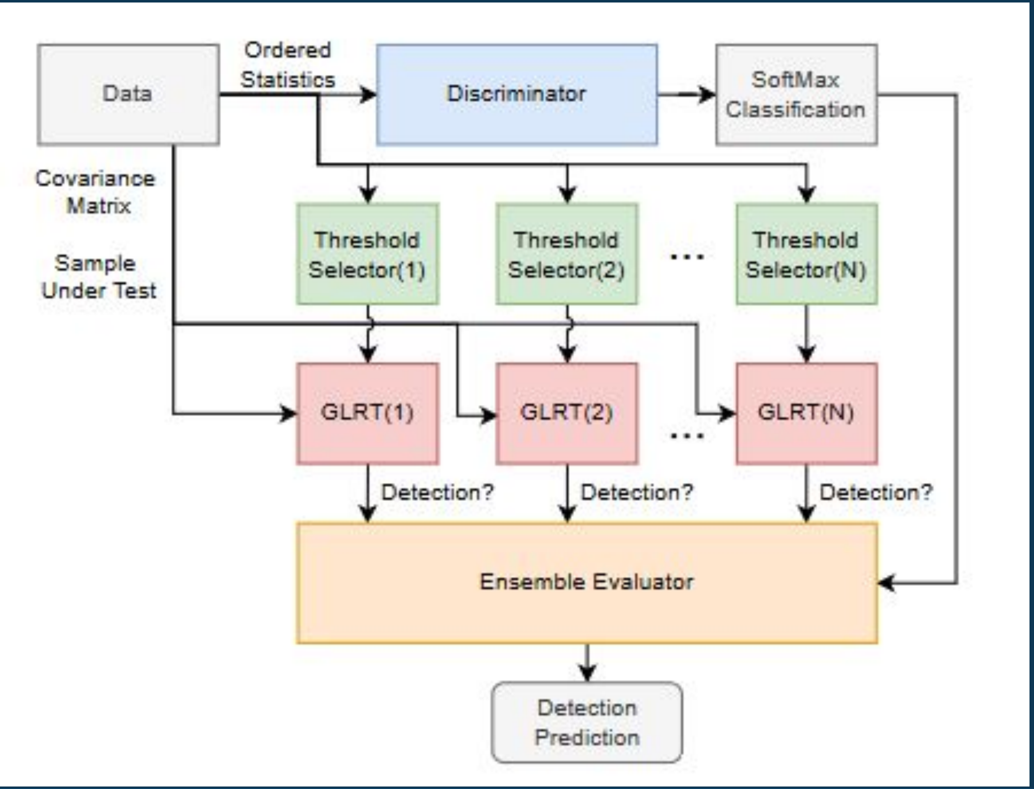
A container must be built before it can be run. This is like compiling code. Afterwards it can be called through the command line or through applications like Kubernetes.



How does this apply to the Dynamic-CFAR algorithm?

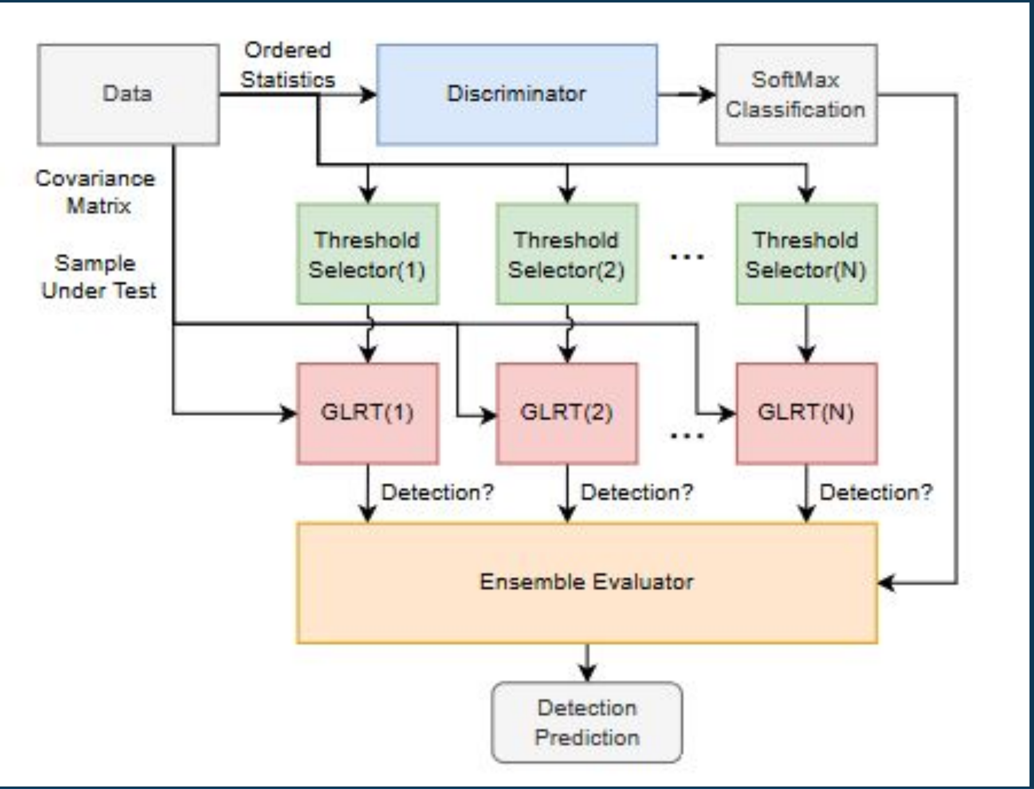


Each segment of the algorithm, including the Discriminator and every unique Prediction Model, can be placed into its own container.



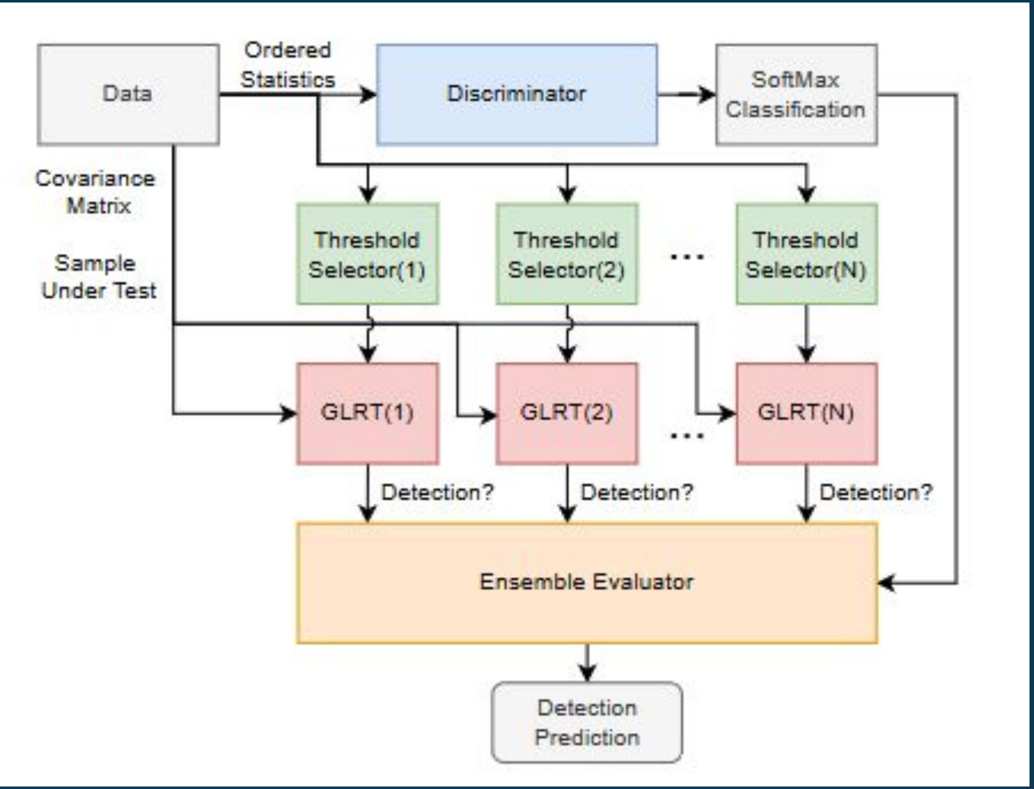
This allows for:

- Increased security via isolated code



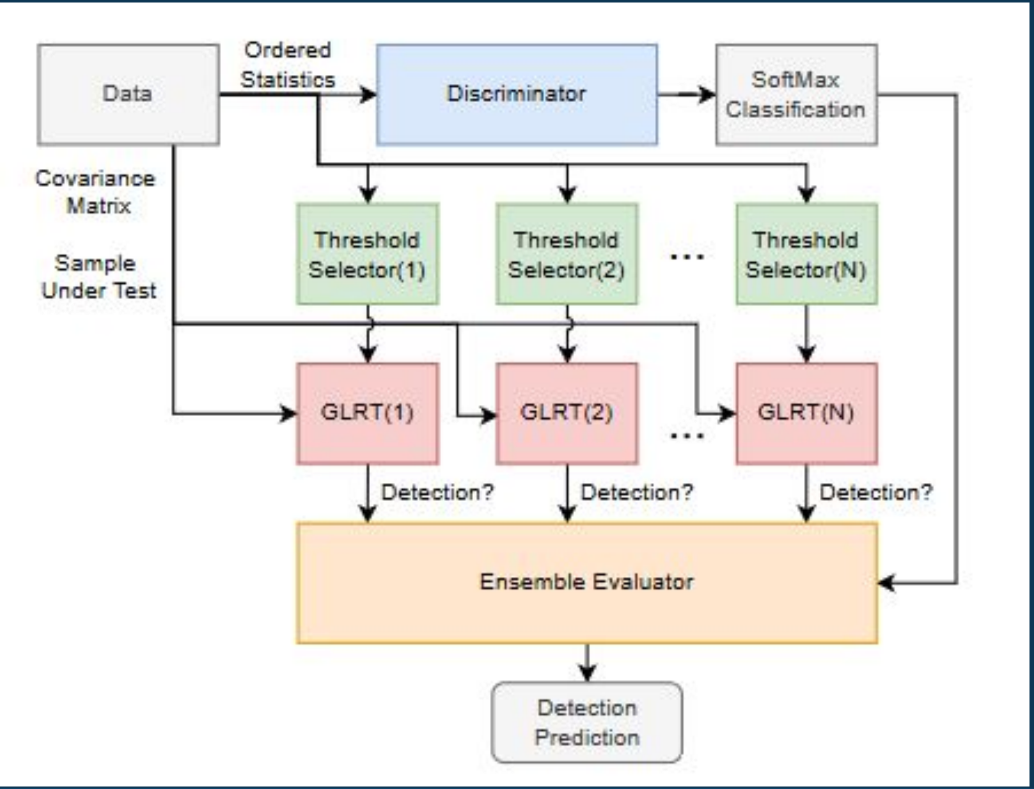
This allows for:

- Increased security via isolated code
- Enhanced versatility via modular architecture with part swapping



This allows for:

- Increased security via isolated code
- Enhanced versatility via modular architecture with part swapping
- Platform independence through Docker



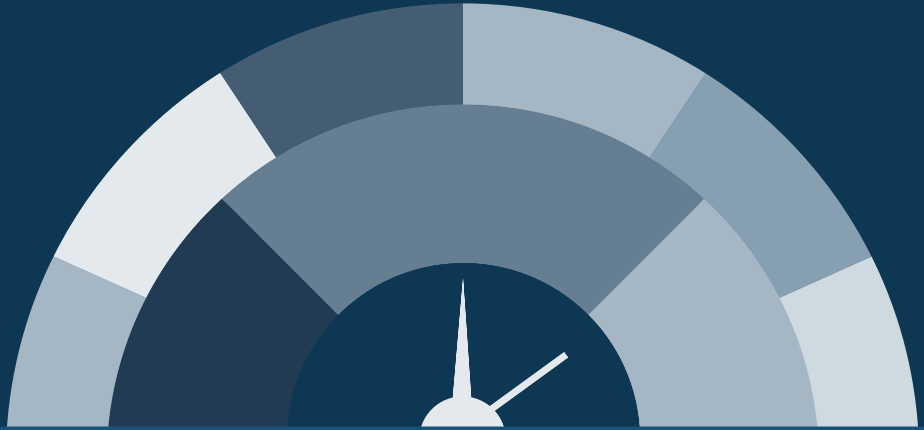
This allows for:

- Increased security via isolated code
- Enhanced versatility via modular architecture with part swapping
- Platform independence through Docker
- Ability to use orchestration tools to build efficient fault tolerant systems

Using Docker Containers, Kubernetes can manage and scale created systems to increase reliability of computing resources.

A few key benefits of Kubernetes:

- Templates can be built to easily replicate container and whole system setup.

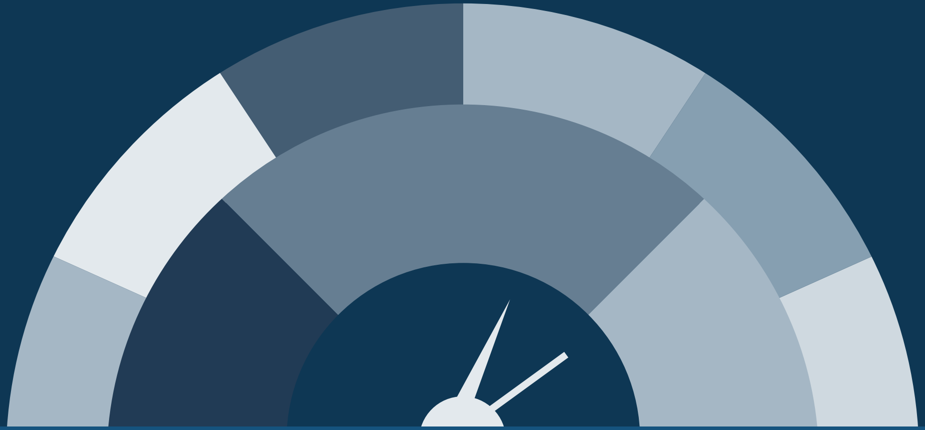


A few key benefits of Kubernetes:

- Templates can be built to easily replicate container and whole system setup.
- Computing resources are monitored and managed automatically.

A few key benefits of Kubernetes:

- Templates can be built to easily replicate container and whole system setup.
- Computing resources are monitored and managed automatically.
- Processes that have failed are automatically stopped, and can be restarted if specified.



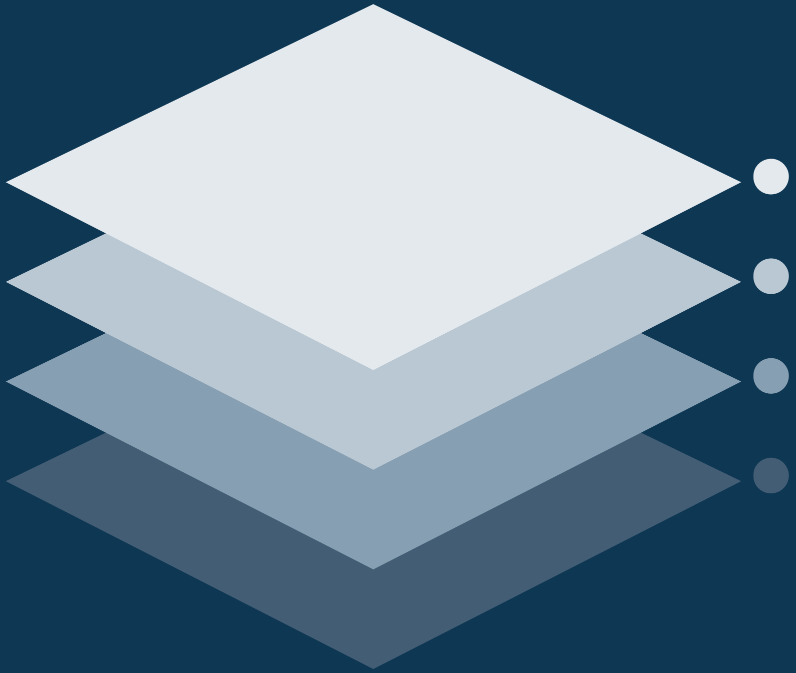
A few key benefits of Kubernetes:

- Templates can be built to easily replicate container and whole system setup.
- Computing resources are monitored and managed automatically.
- Processes that have failed are automatically stopped, and can be restarted if specified.
- Continuous Deployment can be implemented for production applications.

To make the Dynamic-CFAR algorithm a reliable modular system, code components can be isolated within a system of Docker Containers and then deployed with Kubernetes.

02 Code Restructuring

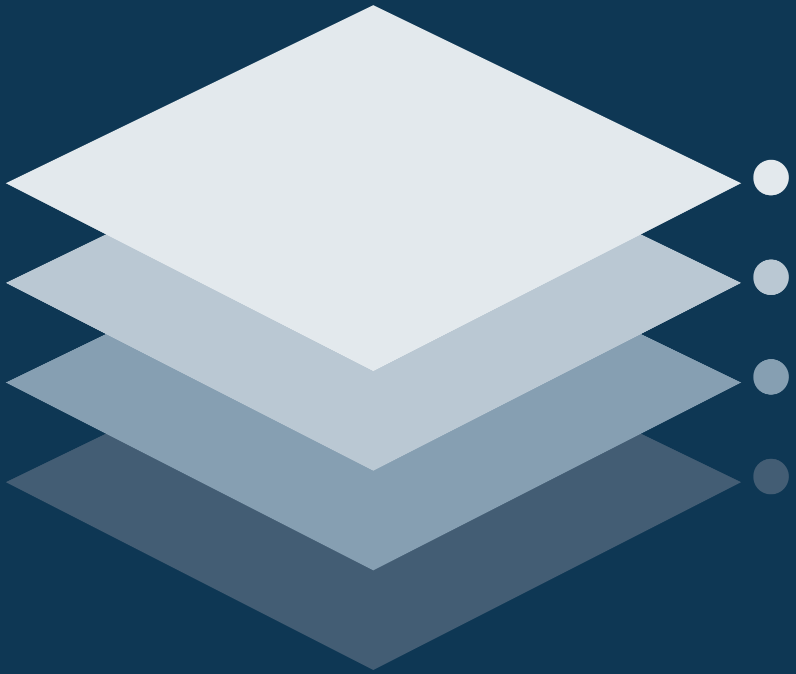
An explanation of changes made to the original Dynamic-CFAR algorithm.



Args_Class_Module.py

To allow the system to run independently, the original command line argument parser has been changed to a separate class module.

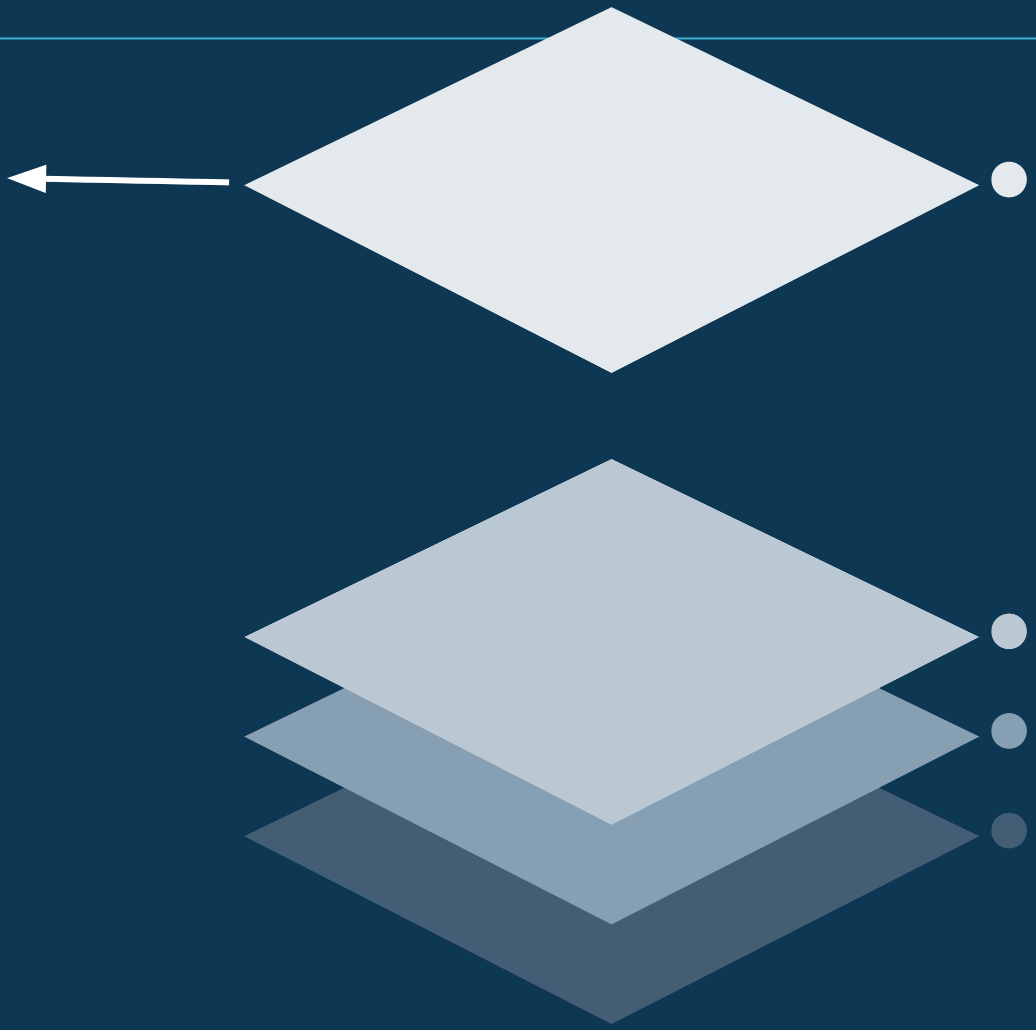
This retains customization by acting as a configuration file for the system.



Docker_Discriminator.py

- First, the data specified in Args_Class_Module.py (Args) is read into a variable.
- The given Discriminator Model is called to interpret the data.
- The raw output of the model is saved into the file: distribution_tensors.csv

- This output is currently the SoftMax probability array for each data point calculating the most likely distribution type for the radar information.



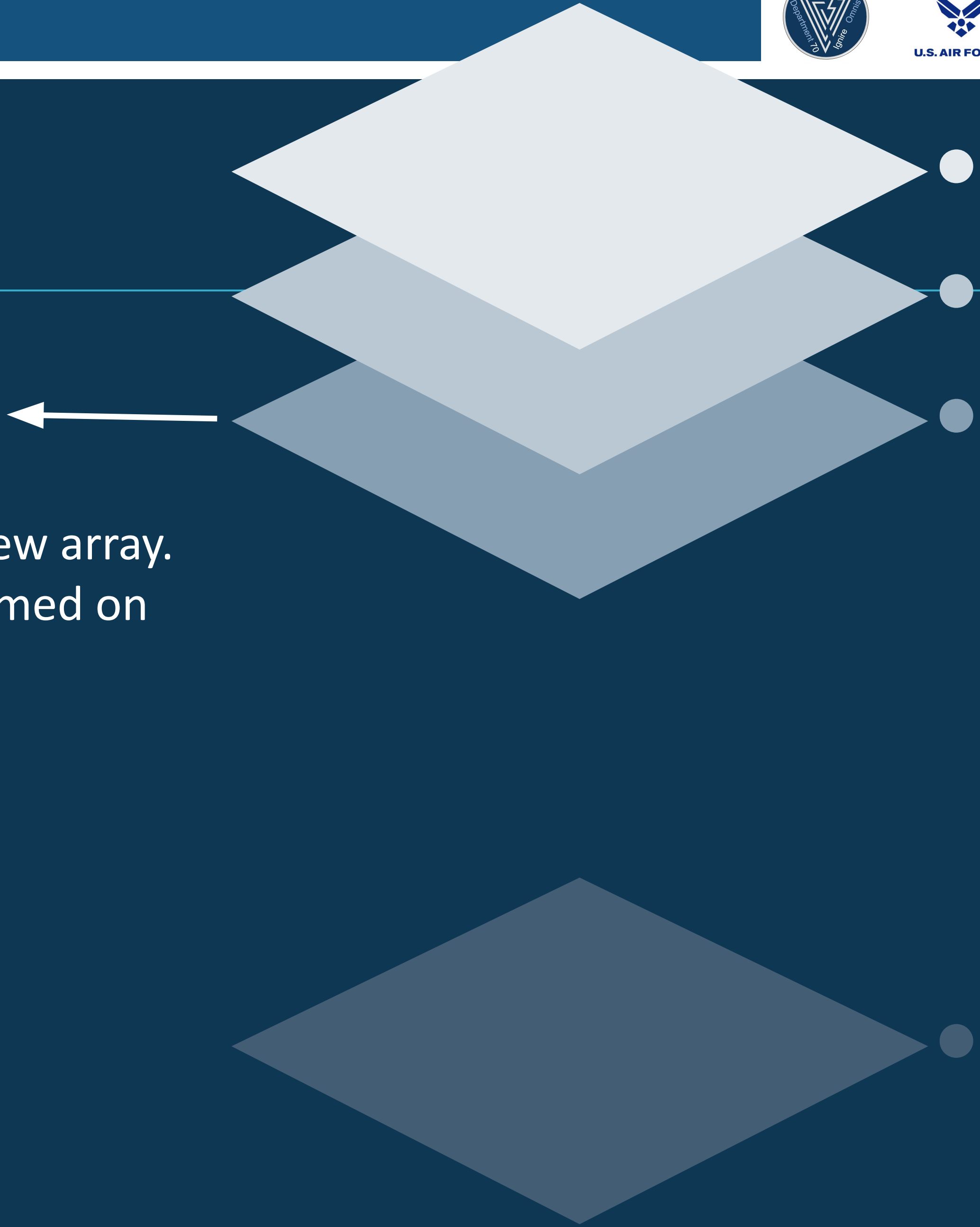
Docker_Evaluator.py

- First, the csv output from the Discriminator is loaded to a new array.

- The given Evaluator Model is called to interpret the data.

- The output of the model is saved into the file: max_disc_list.csv

- This output is currently the ArgMax value of each data point, this corresponds to the distribution type of highest likelihood. Max_disc_list stands for max discriminator list.



Model-N.py

- First, the csv output from the Evaluator is loaded to a new array.
- In each Model Container, a specific calculation is performed on each data point to calculate the number of False Alarms

- The outputs of the models are saved into multiple files:
fa_cd_model_n.csv | fa_glrt.csv | fa_ideal.csv
(GLRT and Ideal are only calculated by one model)

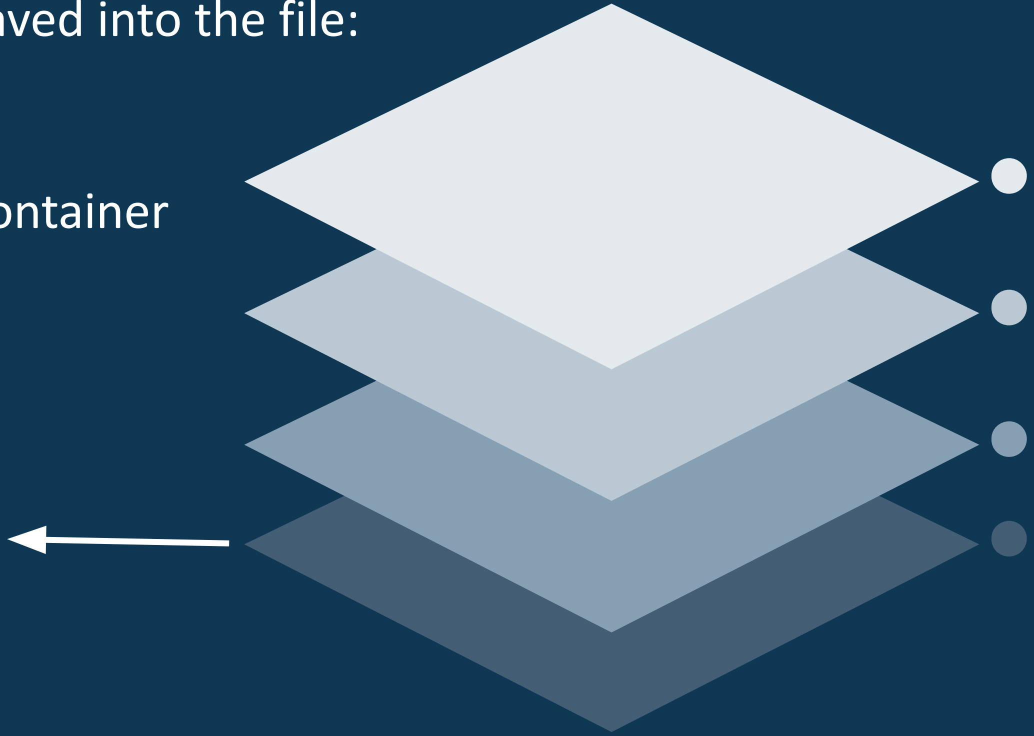
- There are currently seven models in total; however, this number and also the type of Models used for calculations can be changed freely.

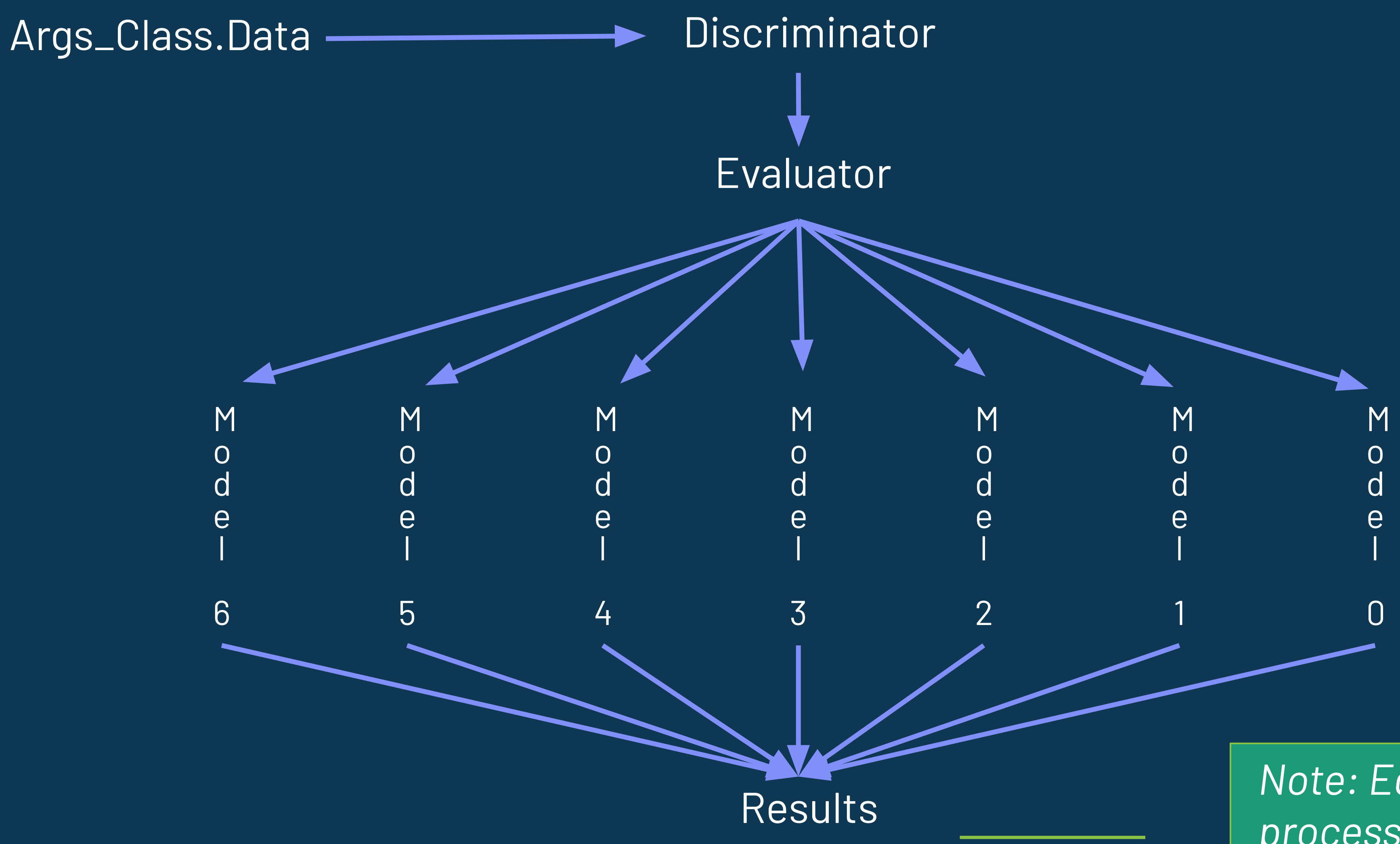
- This container simply sums the results of all of the Model Containers, which corresponds to the number of False Alarms generated.

- The output of the model is printed to the terminal and saved into the file: metacog_results.csv

- If changes to the Models layer are needed, this Results container can easily by updated to reflect the changes made.

Docker_Results.py





Note: Each model does not process all data points, The algorithm is not O(n^7)

For more detailed explanations of the code changes, please view the following file: “ Modular Metacog Code Overview.pdf ”

This can be found in the main project folder.

03 Docker Architecture

A visualization of the created Docker System.

Images → A segment of Dynamic-CFAR code is isolated and packaged into a Docker Image.

Containers → An instance of the code segment can be run in a container.

Volumes → To share data, a Volume File Storage can be mounted to a running container.

base-image

- Each Docker Image for this algorithm needs an operating system in order to run the code segments.

- To increase efficiency, a “Base” Image was created so that the operating system and required Python Libraries would only need to be installed once.

- Instead of installing all of this data in each Image, the following Images simply inherit from the “Base”.

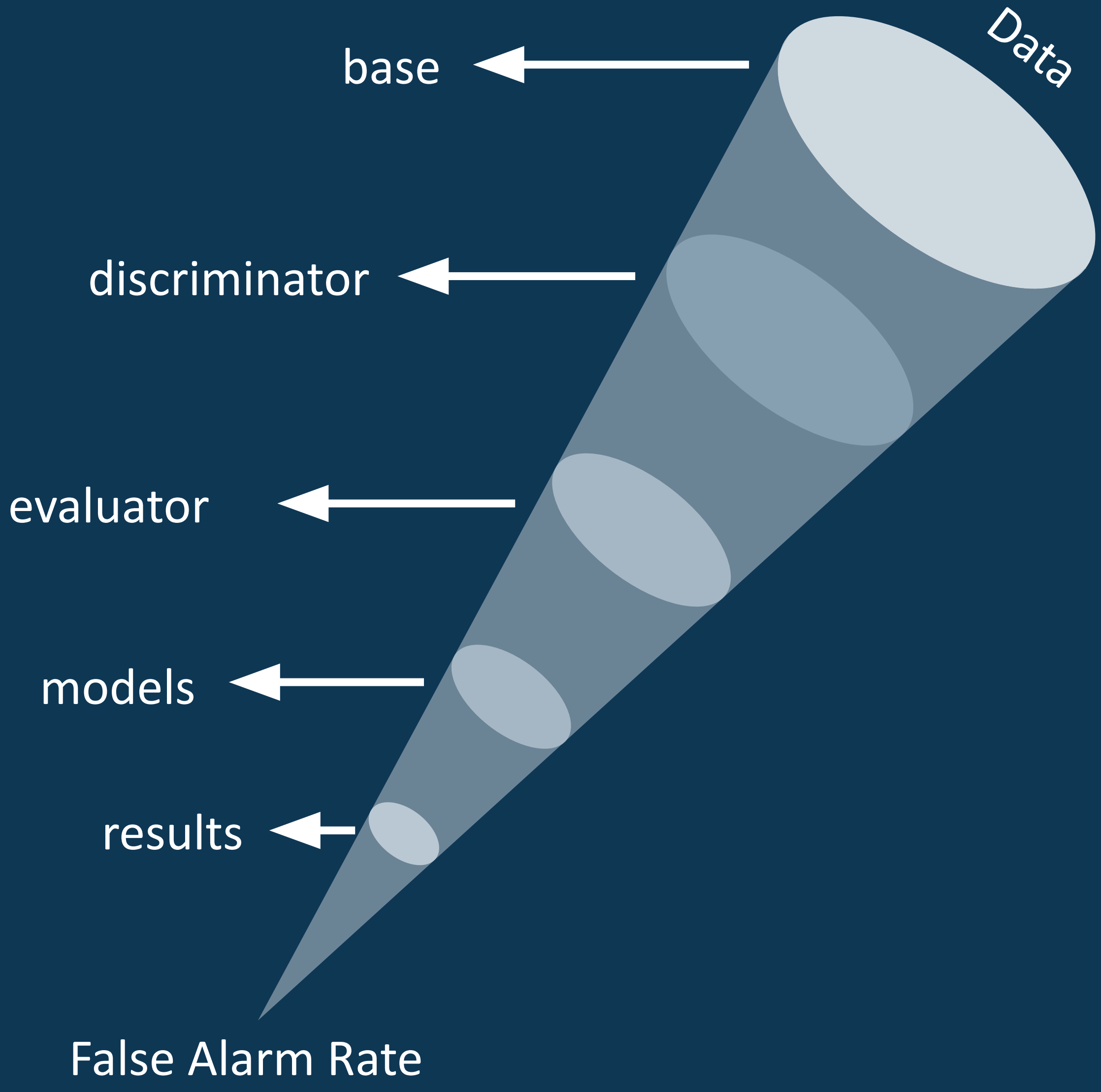
Modular Dynamic-CFAR

- Rather than processing the data with function calls, the Modular implementation filters the data semi-sequentially through the containers.

- Data is shared by saving and loading specific files, unique to each step of the process.

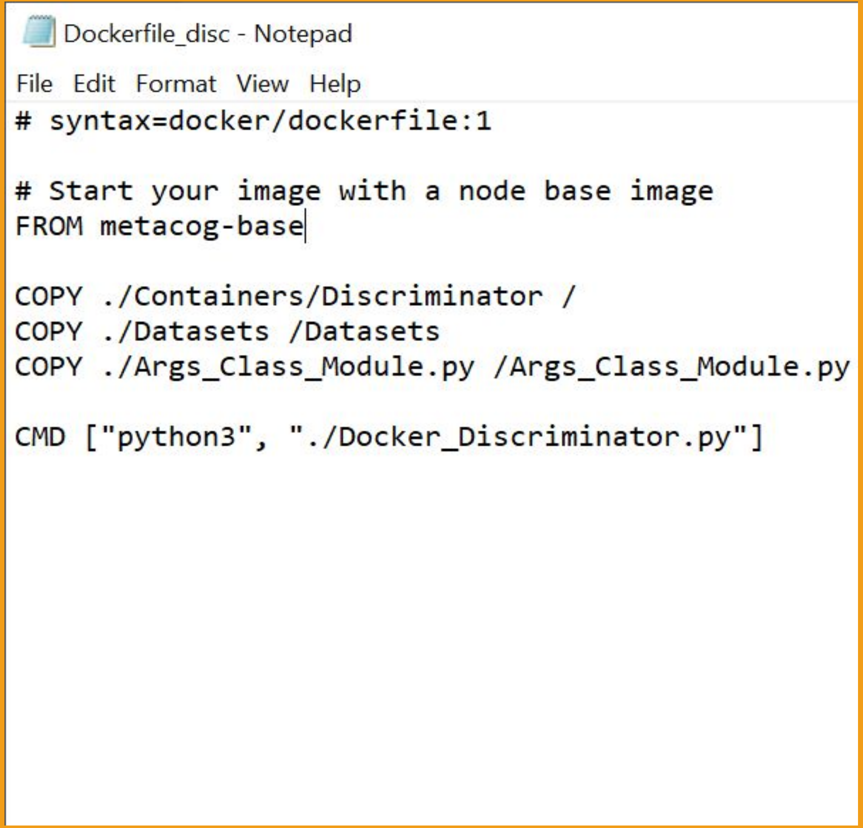
- All containers in the system must be connected to the same Volume to share data.

- Files and code within the Images are not accessible by the host machine.



Each image is configured with a ‘Dockerfile’

FROM	Specifies the operating system to install or base to inherit.
RUN	Runs a command only once, when the Image is built.
COPY	Copies files from the host machine into the Image file system.
CMD	This specifies the command to run every time the Image is instanced in a container. Here, it is the specific code segment of the Dynamic-CFAR algorithm.

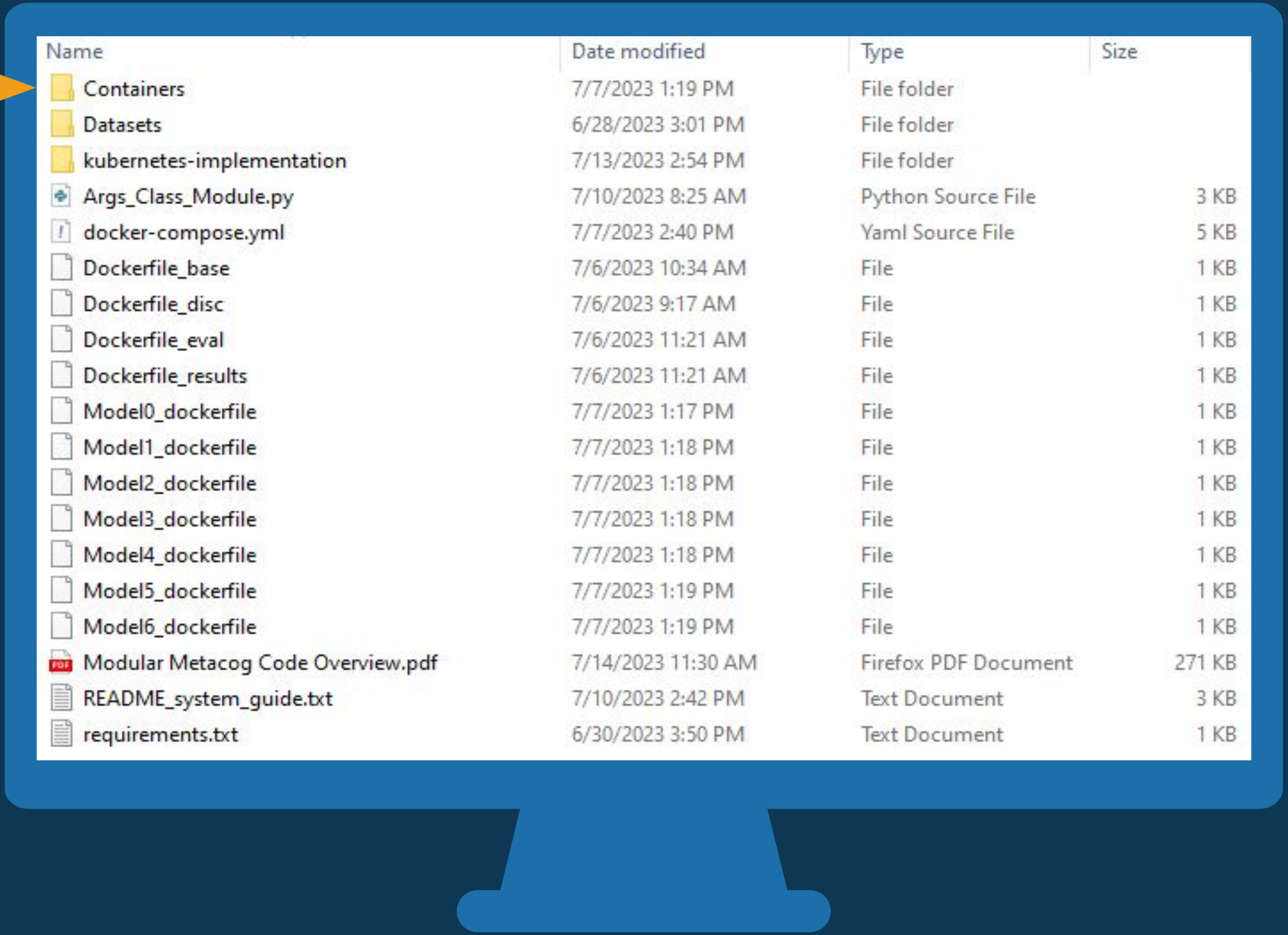


```
# syntax=docker/dockerfile:1

# Start your image with a node base image
FROM metacog-base

COPY ./Containers/Discriminator /
COPY ./Datasets /Datasets
COPY ./Args_Class_Module.py /Args_Class_Module.py

CMD ["python3", "./Docker_Discriminator.py"]
```

Discriminator Example

- FROM : Inherits metacog-base
- COPY : Gets a local copy of the necessary files
- CMD : Runs the Discriminator code segment

(Each Image in the system has a unique Dockerfile that follows this same format)



Name	Date modified	Type	Size
Containers	7/7/2023 1:19 PM	File folder	
Datasets	6/28/2023 3:01 PM	File folder	
kubernetes-implementation	7/13/2023 2:54 PM	File folder	
Args_Class_Module.py	7/10/2023 8:25 AM	Python Source File	3 KB
docker-compose.yml	7/7/2023 2:40 PM	Yaml Source File	5 KB
Dockerfile_base	7/6/2023 10:34 AM	File	1 KB
Dockerfile_disc	7/6/2023 9:17 AM	File	1 KB
Dockerfile_eval	7/6/2023 11:21 AM	File	1 KB
Dockerfile_results	7/6/2023 11:21 AM	File	1 KB
Model0_dockerfile	7/7/2023 1:17 PM	File	1 KB
Model1_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model2_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model3_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model4_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model5_dockerfile	7/7/2023 1:19 PM	File	1 KB
Model6_dockerfile	7/7/2023 1:19 PM	File	1 KB
Modular Metacog Code Overview.pdf	7/14/2023 11:30 AM	Firefox PDF Document	271 KB
README_system_guide.txt	7/10/2023 2:42 PM	Text Document	3 KB
requirements.txt	6/30/2023 3:50 PM	Text Document	1 KB

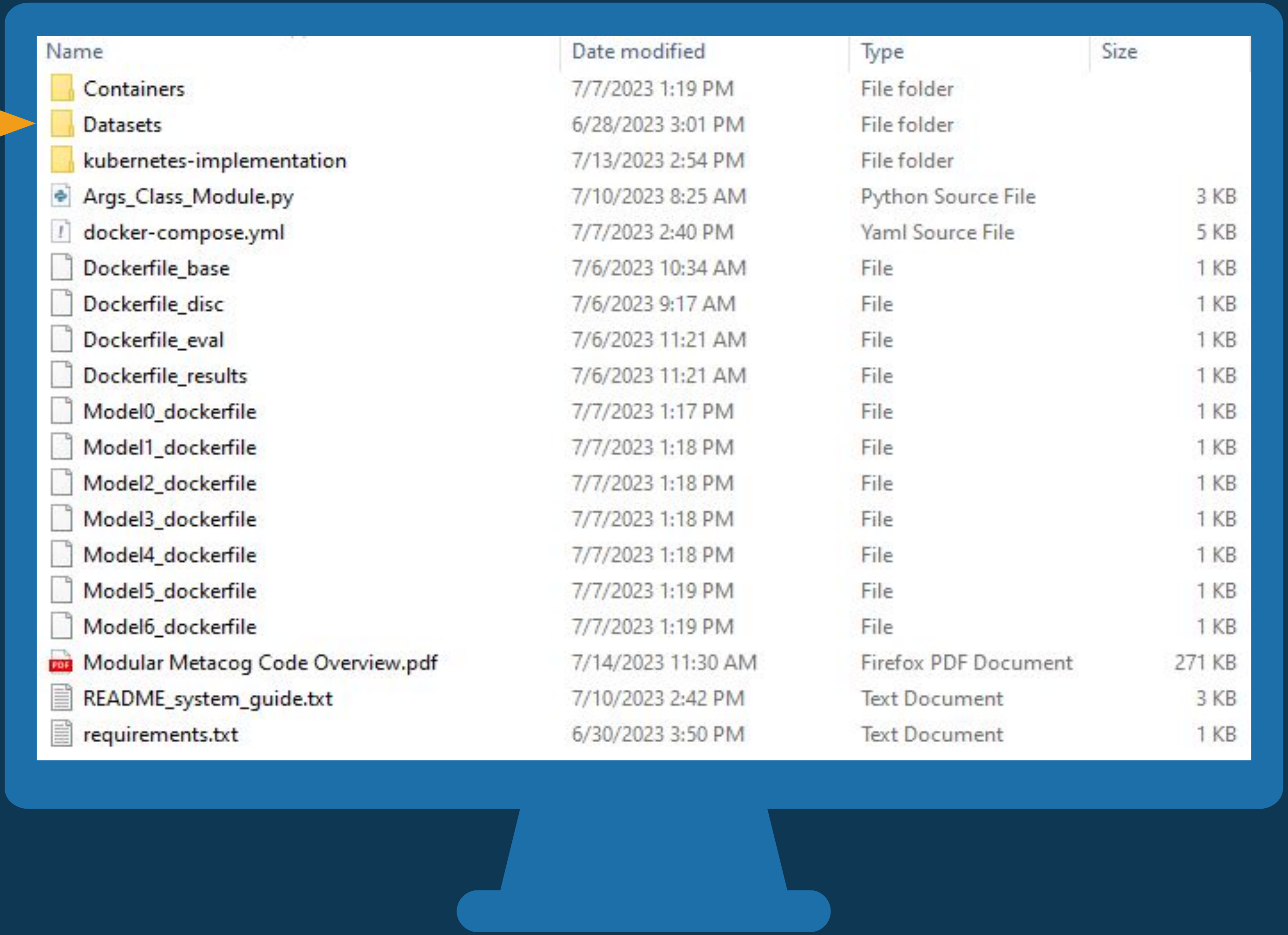
./Containers

This folder contains the files needed for each Image.

- The python script
- The corresponding model

This includes:

- Base, Discriminator Evaluator, Models, and Results



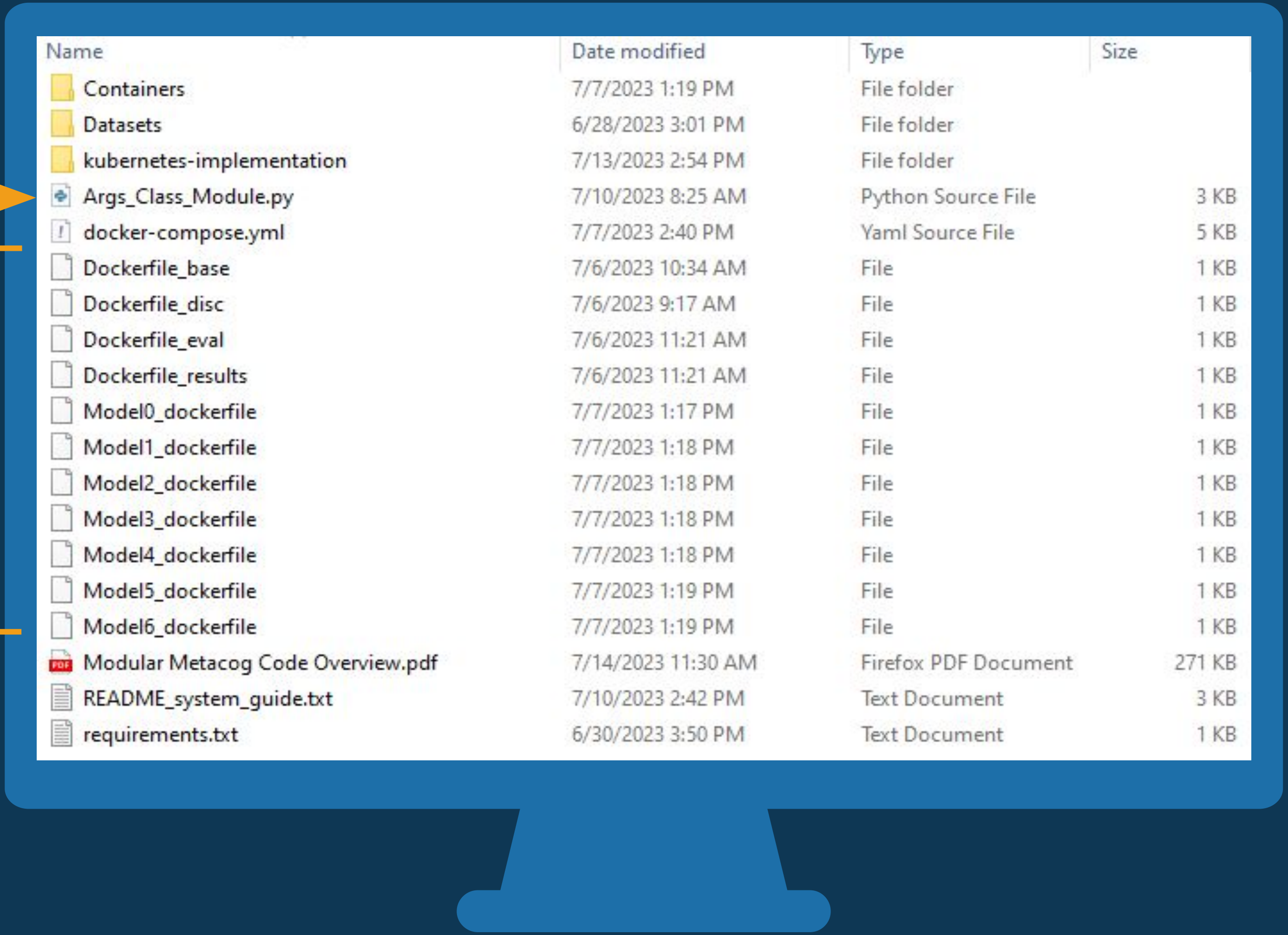
Name	Date modified	Type	Size
Containers	7/7/2023 1:19 PM	File folder	
Datasets	6/28/2023 3:01 PM	File folder	
kubernetes-implementation	7/13/2023 2:54 PM	File folder	
Args_Class_Module.py	7/10/2023 8:25 AM	Python Source File	3 KB
docker-compose.yml	7/7/2023 2:40 PM	Yaml Source File	5 KB
Dockerfile_base	7/6/2023 10:34 AM	File	1 KB
Dockerfile_disc	7/6/2023 9:17 AM	File	1 KB
Dockerfile_eval	7/6/2023 11:21 AM	File	1 KB
Dockerfile_results	7/6/2023 11:21 AM	File	1 KB
Model0_dockerfile	7/7/2023 1:17 PM	File	1 KB
Model1_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model2_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model3_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model4_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model5_dockerfile	7/7/2023 1:19 PM	File	1 KB
Model6_dockerfile	7/7/2023 1:19 PM	File	1 KB
Modular Metacog Code Overview.pdf	7/14/2023 11:30 AM	Firefox PDF Document	271 KB
README_system_guide.txt	7/10/2023 2:42 PM	Text Document	3 KB
requirements.txt	6/30/2023 3:50 PM	Text Document	1 KB

./Datasets

This folder contains the data file that is processed by the Dynamic-CFAR system.

- It is in the main folder not in each Container’s folder.
- It is copied into all Containers, so you can change the file once to update all of the Containers.

./



Name	Date modified	Type	Size
Containers	7/7/2023 1:19 PM	File folder	
Datasets	6/28/2023 3:01 PM	File folder	
kubernetes-implementation	7/13/2023 2:54 PM	File folder	
Args_Class_Module.py	7/10/2023 8:25 AM	Python Source File	3 KB
docker-compose.yml	7/7/2023 2:40 PM	Yaml Source File	5 KB
Dockerfile_base	7/6/2023 10:34 AM	File	1 KB
Dockerfile_disc	7/6/2023 9:17 AM	File	1 KB
Dockerfile_eval	7/6/2023 11:21 AM	File	1 KB
Dockerfile_results	7/6/2023 11:21 AM	File	1 KB
Model0_dockerfile	7/7/2023 1:17 PM	File	1 KB
Model1_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model2_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model3_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model4_dockerfile	7/7/2023 1:18 PM	File	1 KB
Model5_dockerfile	7/7/2023 1:19 PM	File	1 KB
Model6_dockerfile	7/7/2023 1:19 PM	File	1 KB
Modular Metacog Code Overview.pdf	7/14/2023 11:30 AM	Firefox PDF Document	271 KB
README_system_guide.txt	7/10/2023 2:42 PM	Text Document	3 KB
requirements.txt	6/30/2023 3:50 PM	Text Document	1 KB

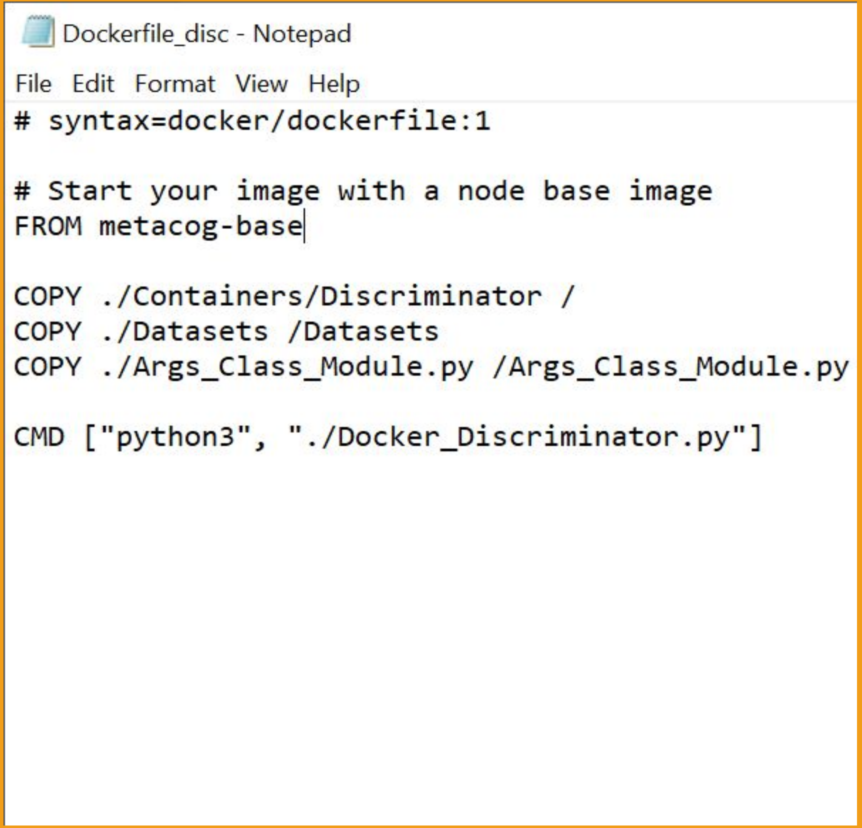
The main directory contains various configuration files.

Args_Class_Module.py

- Inherited by all from a single file like ./Datasets

Dockerfiles

- Each Image is built from the main directory so these are stored in this folder



Dockerfile_disc - Notepad

File Edit Format View Help

```
# syntax=docker/dockerfile:1

# Start your image with a node base image
FROM metacog-base

COPY ./Containers/Discriminator /
COPY ./Datasets /Datasets
COPY ./Args_Class_Module.py /Args_Class_Module.py

CMD ["python3", "./Docker_Discriminator.py"]
```

Discriminator Example

- FROM : Inherits metacog-base
- COPY : Specific Container,
  Common Data File
  Common Args Class File
- CMD : Runs the Discriminator code segment

(Built using the Dockerfile_disc file)

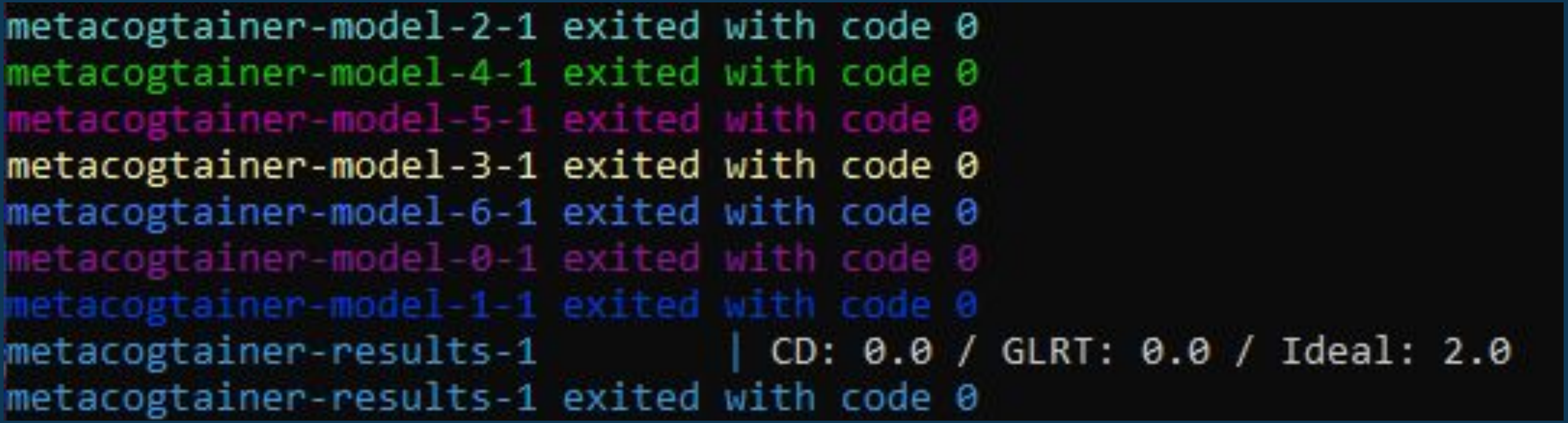
Testing the system

- To run the system navigate to the main directory and use the following command:

  docker compose up

- To close the system use the following command:

  docker compose down

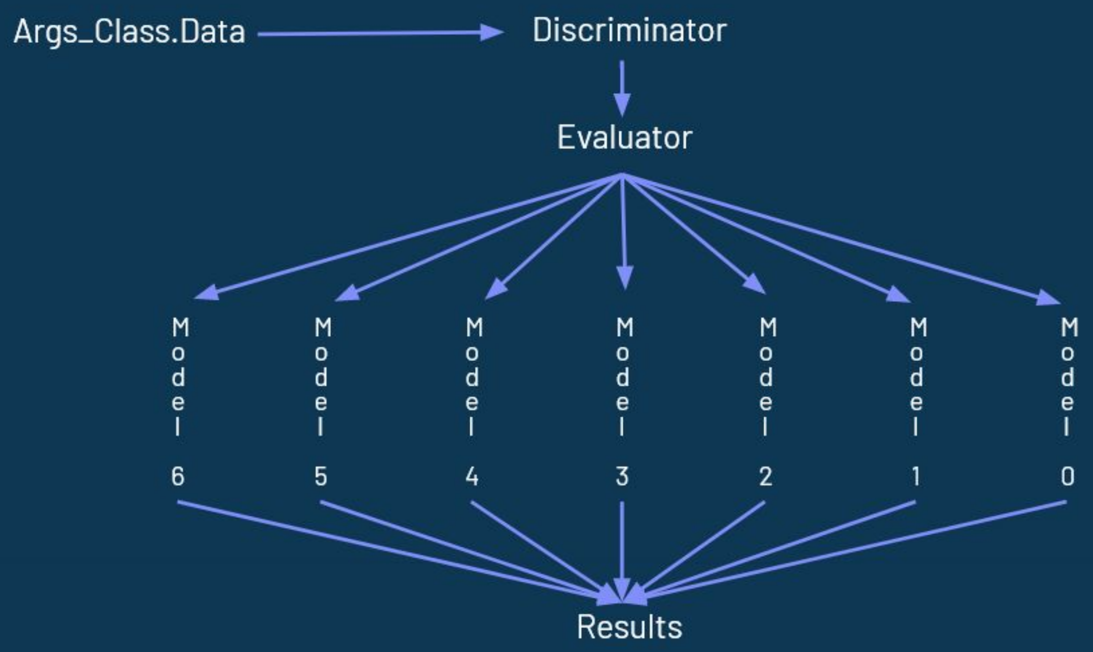


```
metacogtainer-model-2-1 exited with code 0
metacogtainer-model-4-1 exited with code 0
metacogtainer-model-5-1 exited with code 0
metacogtainer-model-3-1 exited with code 0
metacogtainer-model-6-1 exited with code 0
metacogtainer-model-0-1 exited with code 0
metacogtainer-model-1-1 exited with code 0
metacogtainer-results-1        | CD: 0.0 / GLRT: 0.0 / Ideal: 2.0
metacogtainer-results-1 exited with code 0
```

Testing the system

- The Docker Compose function utilizes the " docker-compose.yml " configuration file to build and the run the modular Dynamic-CFAR implementation.

- The Base, Discriminator, and Evaluator run sequentially.
- This is followed by all N Models running in parallel once the evaluator has finished.
- And finally, the Results are gathered once all N Models have finished running.



Docker Limitations

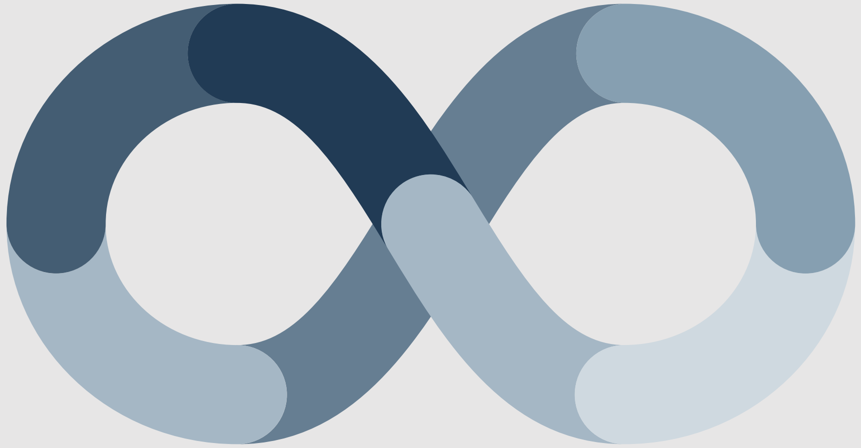
- Using Docker Compose, the system is not continuously available as it closes once it has been run.

- It also does not offer fault tolerance.

- Thus, Kubernetes can be used to further enhance the reliability of the modular system.

04 Kubernetes Implementation

A final proof of concept for secure and reliable modular algorithms.

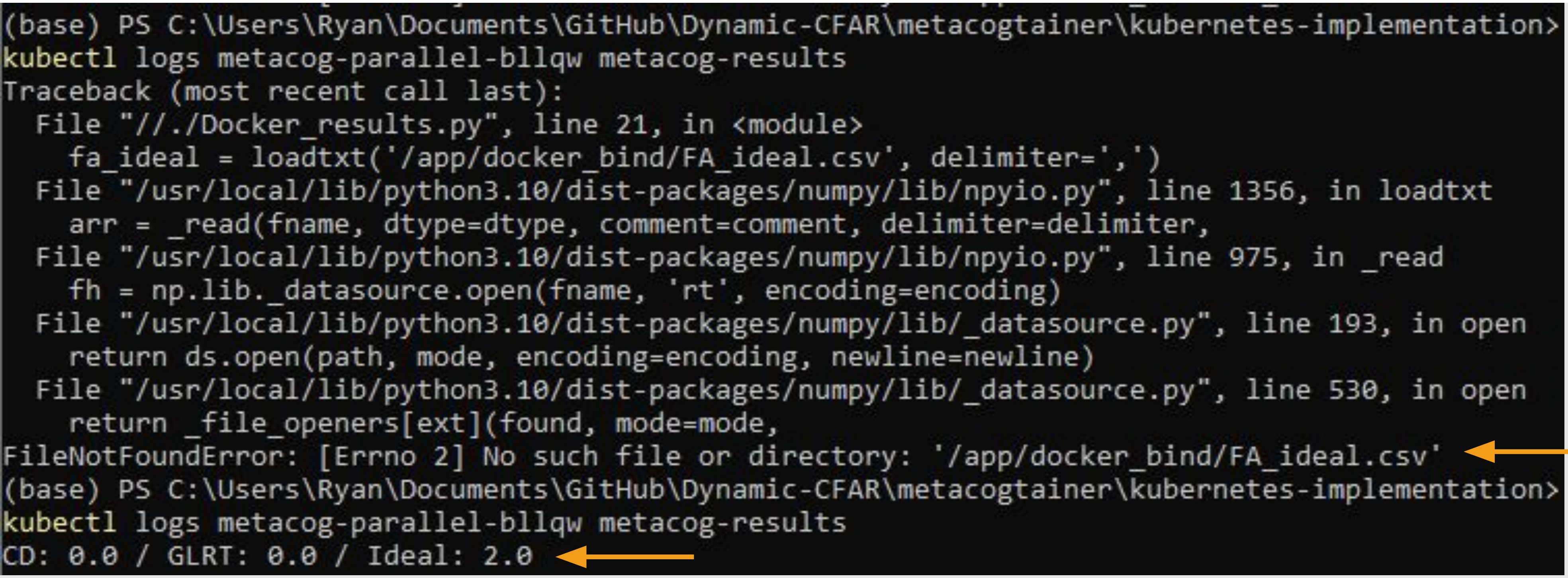


- Kubernetes uses Containers to manage and deploy applications.

- The modular implementation of the Dynamic-CFAR algorithm uses Docker Containers.

- Kubernetes can be applied to enhance the Docker system to add fault tolerance

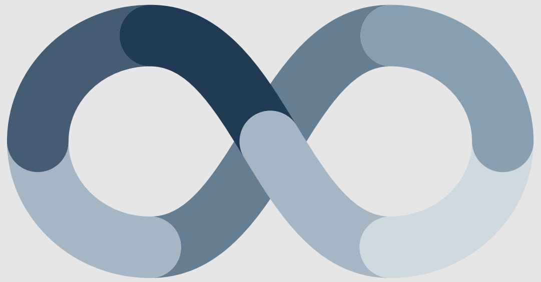
- When an error occurs, only a container would fail, and with Kubernetes this would not affect the results of the systems



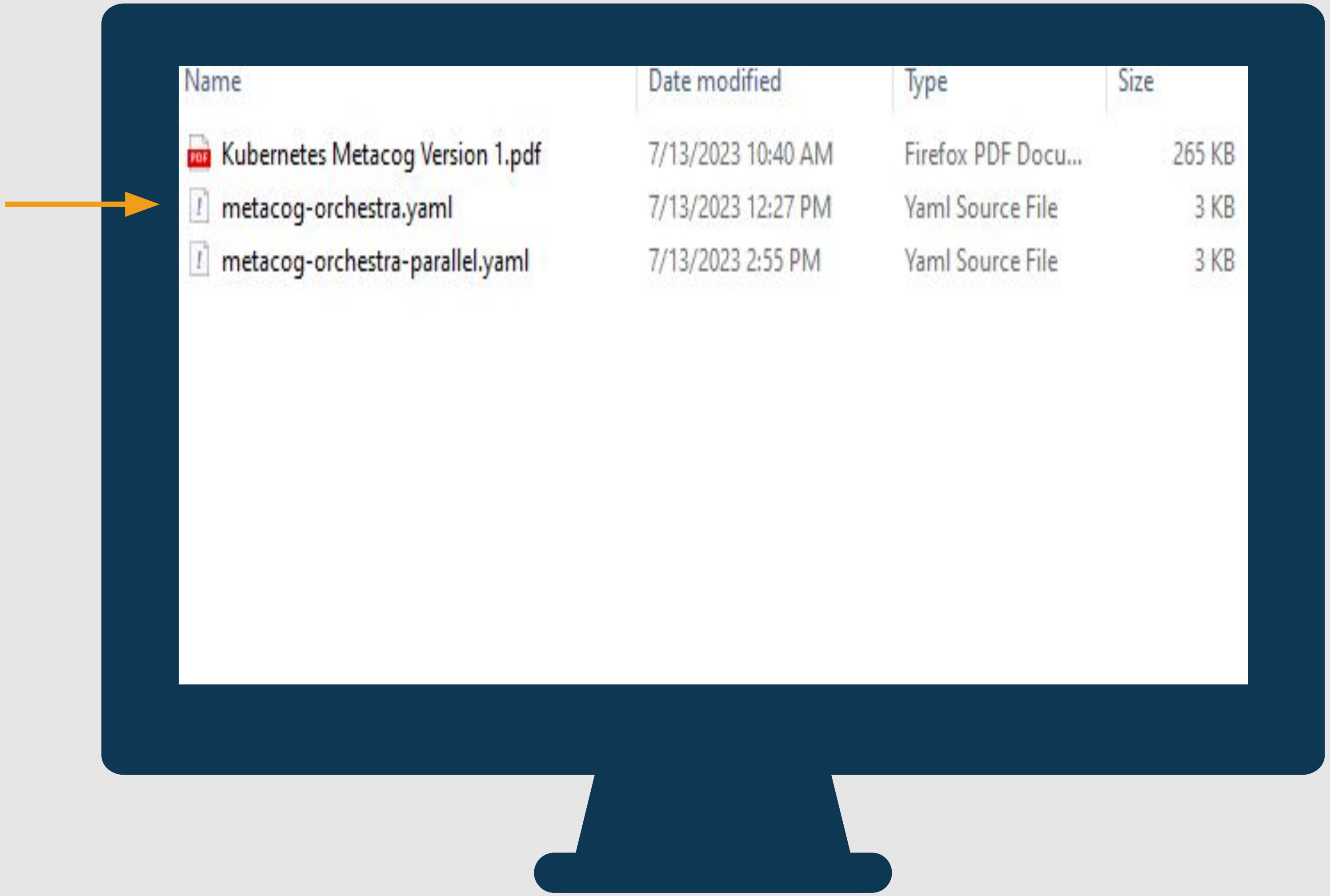
```
(base) PS C:\Users\Ryan\Documents\GitHub\Dynamic-CFAR\metacogtainer\kubernetes-implementation>
kubectl logs metacog-parallel-bllqw metacog-results
Traceback (most recent call last):
  File "//./Docker_results.py", line 21, in <module>
    fa_ideal = loadtxt('/app/docker_bind/FA_ideal.csv', delimiter=',')
  File "/usr/local/lib/python3.10/dist-packages/numpy/lib/npyio.py", line 1356, in loadtxt
    arr = _read(fname, dtype=dtype, comment=comment, delimiter=delimiter,
  File "/usr/local/lib/python3.10/dist-packages/numpy/lib/npyio.py", line 975, in _read
    fh = np.lib._datasource.open(fname, 'rt', encoding=encoding)
  File "/usr/local/lib/python3.10/dist-packages/numpy/lib/_datasource.py", line 193, in open
    return ds.open(path, mode, encoding=encoding, newline=newline)
  File "/usr/local/lib/python3.10/dist-packages/numpy/lib/_datasource.py", line 530, in open
    return _file_openers[ext](found, mode=mode,
FileNotFoundError: [Errno 2] No such file or directory: '/app/docker_bind/FA_ideal.csv'
(base) PS C:\Users\Ryan\Documents\GitHub\Dynamic-CFAR\metacogtainer\kubernetes-implementation>
kubectl logs metacog-parallel-bllqw metacog-results
CD: 0.0 / GLRT: 0.0 / Ideal: 2.0
```

- In this example, an error is induced on purpose. The Results container tries to gather data before the Models have finished running.

- However, with Kubernetes' system health checks, the system does not fail. Instead it waits until the Models have run and finishes successfully.



./kubernetes-implementation



Name	Date modified	Type	Size
Kubernetes Metacog Version 1.pdf	7/13/2023 10:40 AM	Firefox PDF Docu...	265 KB
metacog-orchestra.yaml	7/13/2023 12:27 PM	Yaml Source File	3 KB
metacog-orchestra-parallel.yaml	7/13/2023 2:55 PM	Yaml Source File	3 KB

- Two Kubernetes applications have been created to manage the modular system of Dynamic-CFAR containers.
- The first file is documentation on the setup of the system.
- The second file is the basic implementation of a Kubernetes Job, this version runs sequentially.
- The third file is the same, but with parallel processing of Model Containers.

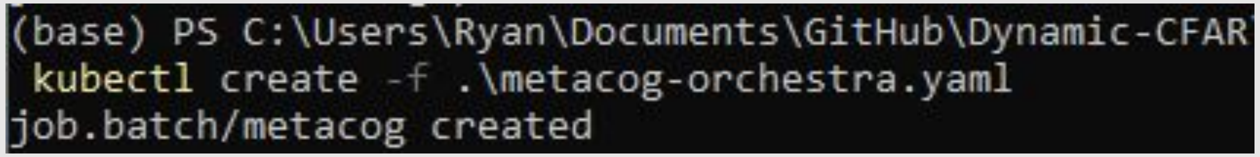
Testing the Kubernetes System

kubectl create -f metacog-orchestra.yaml

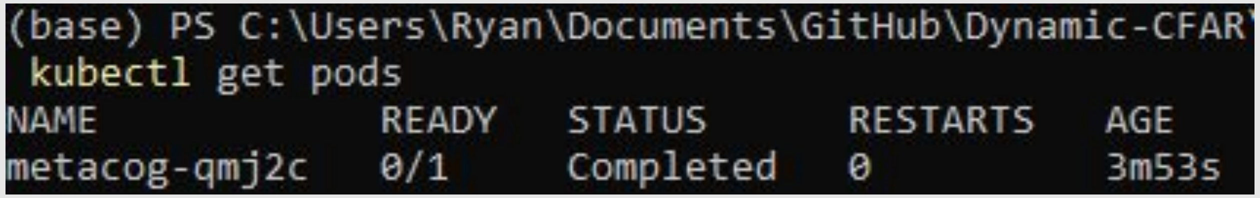
- Creates a “Job”, which is an instance of running the Dynamic-CFAR system.
- The name is currently ‘metacog’, and is specified in the YAML file.

kubectl delete jobs metacog

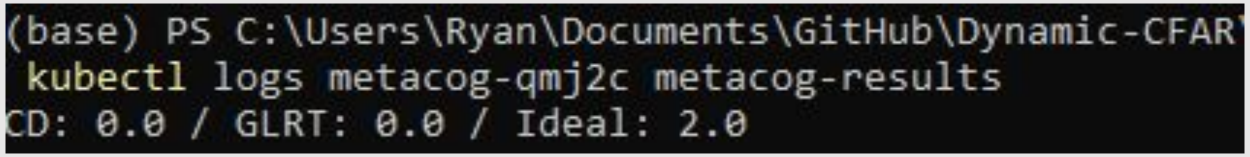
- Deletes (Closes) the active instance.



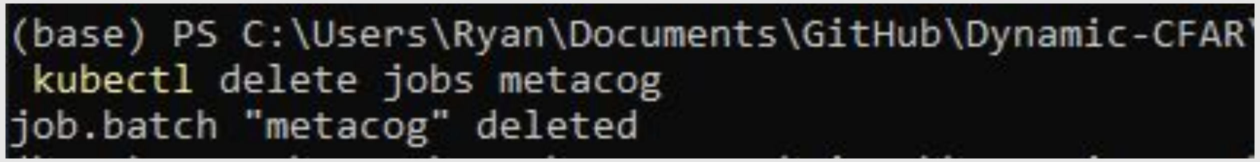
```
(base) PS C:\Users\Ryan\Documents\GitHub\Dynamic-CFAR
 kubectl create -f .\metacog-orchestra.yaml
job.batch/metacog created
```



```
(base) PS C:\Users\Ryan\Documents\GitHub\Dynamic-CFAR
 kubectl get pods
NAME             READY   STATUS      RESTARTS   AGE
metacog-qmj2c    0/1     Completed   0          3m53s
```



```
(base) PS C:\Users\Ryan\Documents\GitHub\Dynamic-CFAR
 kubectl logs metacog-qmj2c metacog-results
CD: 0.0 / GLRT: 0.0 / Ideal: 2.0
```



```
(base) PS C:\Users\Ryan\Documents\GitHub\Dynamic-CFAR
 kubectl delete jobs metacog
job.batch "metacog" deleted
```

Summary

- The Dynamic-CFAR algorithm was first split into code segments and restructured to filter data and share information via saving and loading files.

- The algorithm was made platform independent and modular by packaging the code segments into Docker Containers.

- Finally, the architecture built with containers was made fault tolerant with Kubernetes.

Current Research Understanding

Docker :

- Easier to implement and test a system
- Can specify direct dependencies between code segments
- Parts of a system can be run independently for debugging

Kubernetes:

- More advanced capabilities
- Direct dependencies can only be implemented with addons like Argo Workflows
- Applications can be continuously run
- Applications have system health checks and load balancing

Potential Future Applications

- Local Web / GUI interface for interpreting results
- Continuous deployment for continuous input
- Local Web / GUI interface for changing system architecture to fit situational need