ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 01 January 2024

Architectural shift in compute environment for Satellite data processing

[1] Gaurav Gupta, [2] Pradeep C, [3] Murali Krishna ANSV, [4] G Prasad, [5] Manju Sarma

[1] [2] [3] [4] [5] National Remote Sensing Centre, ISRO

Corresponding Author Email: [1] gaurav_gupta@nrsc.gov.in, [2] pradeep_c@nrsc.gov.in, [3] muralikrishna_ansv@nrsc.gov.in, [4] prasad_g@nrsc.gov.in, [5] manjusarma_s@nrsc.gov.in

Abstract— *In the present era of rapid software development, selecting the appropriate architectural approach is important in building flexible, reliable, scalable and maintainable applications. The prominent architecture prevalent primarily in recent years is bare metal compute infrastructure. This paper aims to understand the monolithic approach developed in NRSC and proposes a new approach for data acquisition, processing and work flow modules by instituting a containerized environment and various level 2 and 3 microservices in air-gapped environment in contrast to traditional monolithic approach. Resource utilization of systems in monolithic architecture and in containerized environment with statistics of improved Turn-around Time of processes running in container orchestrated platform in contrast to monolithic architecture is presented in the paper. Meeting the goals of establishing and maintaining a high available and load balanced containerized environment using customized configuration in air gapped network constitute a significant challenge.*

Index Terms— *ADP, API, DPWFM, FRED, IMGEOS.*

I. INTRODUCTION

National Remote Sensing Centre (NRSC) is the premier organization, under ISRO, Dept. of Space which is involved in ground segment activities related to Remote Sensing Satellite missions. NRSC is primarily responsible for the data reception, data product generation and data dissemination. These operations are carried out using the bare metal computers with different configurations.The three major processing elements are Data Acquisition, Data Processing and Data Dissemination as shown in Figure 1 and this segment explains about the existing IT Infrastructure which is monolithic in deployment and the challenges involved in managing IT Infrastructure.

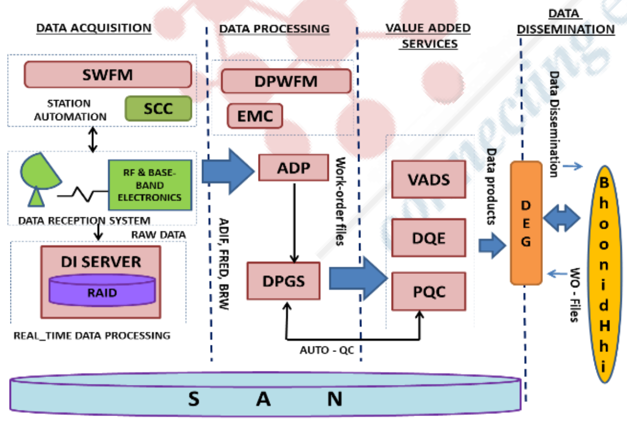


Figure 1. IMGEOS Workflow Chain

Compute, storage and network infrastructure at IMGEOS was installed and integrated as a typical on-premises solution for organization's requirements. The Station Work Flow Manager system starts with the scheduling of ground station sub-systems for all satellite orbits at all the ground stations maintained by NRSC from the Pass Programming System (PPS) on a daily basis. The Data Ingest system on receiving a work-order from SWFM, acquires satellite data and ingests the RAW data at IMGEOS ground station into the shared storage path and updates the data ingest status to the SWFM controller daily. To carry out Data Ingest eight servers are configured. The ADP system performs level-0 processing and generates Orbit Attitude and Time (OAT) File, Framed RAW Extended Data (FRED) and Ancillary Data Information File (ADIF) and twelve servers are configured.

The level 0 processed output is fed as input to the Data Products Generation System (DPGS) to carryout advanced levels of Image processing and data product generation. To carry out the operations of DPGS there is a main controller system known as Data Processing Workflow Manager (DPWFM). The DPWFM floats the work-order files into the data production chain by passing the segregated work-orders to the respective master DP scheduler configured in Master-Slave configuration. The finished products are then subjected to quality check before dissemination. To carry out processing twenty five servers are configured. In total around hundred servers are configured and maintained to support all the operations of IMGEOS chain. The utilization of the available infrastructure is the key measure to work on resource optimization. To optimize the resources at IMGEOS a solution to build containers is proposed in this paper that improves the resource utilization and also simplifies manageability of these huge numbers of systems. Containerization of ground segment data processing software will lead to optimal utilization of infrastructure, better resource management and application deployment. To deploy containerized applications, Kubernetes cluster is built in air gapped test environment at IMGEOS.

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

Volume 11 Issue 01 January 2024

Significant differences of the two mainstream system architecture approaches i.e. monolithic and mircoservices architecture, investigates the resource utilization while using monolithic approach in IMGEOS and determines efficient containerized system architecture suitable for IMGEOS. An open source kubernetes orchestration platform is established in a test network at IMGEOS with load balancer and high availability feature. Kubernetes is a portable, extensible, open source container orchestration platform for managing containerized workloads and services designing a robust air-gapped system, identifying essential and optional component, and the processes needed to operate it. The paper also presents the architecture of resulting Kubernetes infrastructure, discusses the various design decisions and challenges encountered as well as touches upon the processes complementing the system set-up [1]. Building this offline orchestration environment will enable the application developers to host their microservices in a robust, Load balanced and high available environment within IMGEOS data network. Various developer favorable features such as horizontal/vertical pod auto scaling, resource alerting and visualization, tracking resource metrics statistics, etc. have been introduced to the system

II. MICRO SERVICE DEPLOYMENT IN IMGEOS

There are various container orchestration platforms available online in which Kubernetes is chosen and implemented in a test network at IMGEOS. The microservices architecture is method which breaks down a software application into a collection of smaller independent services that communicate over well-defined light-weight APIs [2]. Such architecture are agile and allows application to scale easily and develop faster. Each service covers its own scope and has specific functionality. Such services can be independently deployed, scaled and updated and maintained as per user requirement for a specific function.

Thus the software running in different domains such as ADP, DPS and PQC are proposed to run in containerized microservices system. Such architecture will allow the servers to work as shared pool of resources which will permit different work centers process to run on same sever thus optimizing the resource utilization and make efficient use of infrastructure. A Kubernetes cluster consists of a set of worker machines, called nodes that run containerized applications was built. Every cluster has at least one worker node. The worker node(s) host the pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability. Three high end servers are incorporated viz. two systems with 32 cores CPU and 512 GB RAM, one system with 16 core CPU and 128 GB RAM. Upon these machines seven VMs were built with three master nodes, two Load balancers VMs for high availability and 2 worker node VMS. Kubernetes brings greater reliability and stability to the container-based distributed application, through the use of dynamic scheduling of containers. But Kubernetes cluster should stay up when a component or its master node goes down. Kubernetes High-Availability is about setting up Kubernetes, along with its supporting components in a way that there is no single point of failure. In a single master cluster the important component like API server, controller manager lies only on the single master node and if it fails one cannot create more services, pods etc. However, in case of Kubernetes HA environment, these important components are replicated on multiple masters (three masters in IMGEOS environment) and if any of the masters fail, the other masters keep the cluster up and running. Three master nodes working in active–active configuration are configured as shown in Figure 2. By providing redundancy, a multi-master cluster serves a high available system for end user [3]. To provide load balancing for Linux system and Linux based infrastructure keepalived is used. Keepalived implements a set of checkers to dynamically and adaptively maintain and manage a load-balanced server pool according to their health. Two RHEL VMs are created to work as Load Balancer. Over each Load balancer VM a keepalived daemon and a HAproxy daemon runs which provides uses VRRP (Virtual Router redundancy Protocol) protocol creating virtual routers that bind to a floating, virtual IP address that can be shared between an active and standby HAProxy instance.

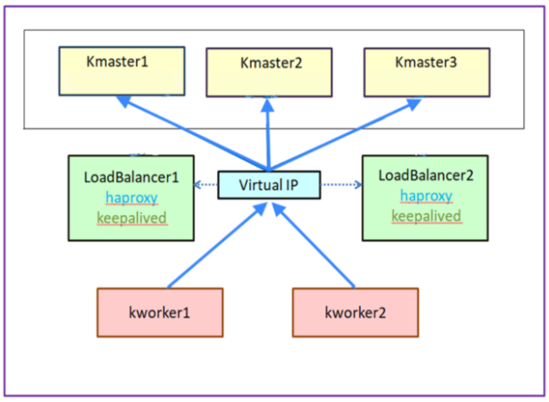


Figure 2. Kubernetes Architecture in IMGEOS

III. CHALLENGES ENCOUNTERED DURING IMPLEMENTATION.

Redhat enterprise virtualization platform which uses Redhat Enterprise Linux and KVMs has been used. Built networking bridge for VM-base machine communication. After literature survey and multiple iterations of testing and evaluation RHEL 7.5 and above Operating System was decided as suitable for Kubernetes installation in airgapped environment.

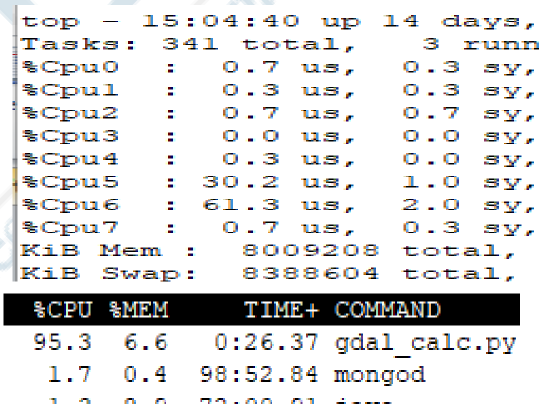
There are various open source container runtime engine available such as docker, containerd, runc, CRI-O, etc. to run container application. Initially docker was opted as it's a popular open source tool in DevOps and provides large community support for bugs and errors. After exhaustive testing and assessment of various packages and features, it was observed that docker does not extend support to latest versions of Kubernetes orchestration platform features and versions. Thus containerd was inculcated as the container runtime engine. But because of lack of community support and one stop solution, various difficulties were faced while installing the runtime engine. Configuration specific to air-gapped environment was carried out. The runtime service (runc) was customized as per the proper version and networking of the Kubernetes architecture in offline environment. Specific containerd configuration plugins were selected to facilitate setting up container runtime over the host OS in all the cluster nodes.

Kubernetes supports Container Networking Interface plugins for cluster networking. Initially Flannel plugin was chosen and thoroughly tested in offline environment. But after configuring it, the underlying container runtime failed to load the CNI plugin required to implement the Kubernetes network. Similarly weave CNI was also tested but containerd runtime communication with Kubernetes using the plugin was not established. Calico CNI, an open source networking and network security solution for Kubernetes was finally added. It provides highly efficient pod networking which uses overlay networking options for air gapped application deployments. Three calico components were added 1.felix which provides routes and ACLs, and other desired connectivity for the endpoints on host. 2.bird it distributes Routing information to peers on the network for inter-host routing using BGP protocol and.3.confd it monitors calico datastore and dynamically creates BIRD configuration files as per updates in datastore.

Kubernetes DNS service is required for service discovery within the cluster. Additional Kubernetes plugin coreDNS was configured to handle all queries in local cluster zone and connect to Kubernetes in-cluster. Customised coreDNS plugins were used to enable service discovery. To enable IMGEOS SAN file system access to the pods, file systems were configured on the worker nodes [4]. Customised deployment workloads were created to mount SAN file system to the pod. Rigorous I/O operations were conducted on file system using pod container and thorough testing was carried. Pods were successfully able to read and write on the file system.

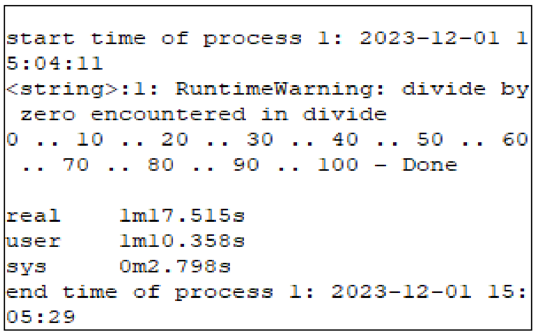
IV. TESTING AND PERFORMANCE COMPARISON

To analyze the working of monolithic architecture, an 8 core virtual machine with 8GB RAM was configured with an opensource geospatial data manipulation software library GDAL which is used for handling raster and vector geospatial data formats. Few basic arithmetic operations were performed by using the gdal_calc library and the corresponding resource utilization was captured. Single instance of the process as well as multiple parallel instances were executed and it was observed that turn around time (TAT) for single instance is 1min 13sec consuming 1 core (12.5%) of CPU and 540 MB of Memory (Figure 3). Running 8 parallel instances of same process resulted in average TAT of 1 min 16 sec as all 8 cores (12.5% CPU per process) were utilized uniformly consuming 540 MB RAM on an average (Figure 4). Till this point the system allocates resources to the processes such that each process can consume memory and CPU to its maximum potential leading to minimal TAT. But when the workload was increased and 14 parallel instances were executed, the average TAT increased to 7 min 38 sec with average CPU utilization of 0.4 core (5.2%) and 516 MB of RAM (Figure 5). Thus it is concluded that when the workload is increased beyond a threshold, due to limited resources the process TAT gets affected. Additionally, it is observed that the resources of peer systems are idle but they cannot be utilized as there is no mechanism for resource sharing. This leads to inefficient resource utilization of systems in the infrastructure.



```
top - 15:04:40 up 14 days,
Tasks: 341 total,   3 runn
%Cpu0  :  0.7 us,  0.3 sy,
%Cpu1  :  0.3 us,  0.3 sy,
%Cpu2  :  0.7 us,  0.7 sy,
%Cpu3  :  0.0 us,  0.0 sy,
%Cpu4  :  0.3 us,  0.0 sy,
%Cpu5  : 30.2 us,  1.0 sy,
%Cpu6  : 61.3 us,  2.0 sy,
%Cpu7  :  0.7 us,  0.3 sy,
KiB Mem :  8009208 total,
KiB Swap:  8388604 total,
 %CPU %MEM     TIME+ COMMAND
 95.3  6.6   0:26.37 gdal_calc.py
  1.7  0.4  98:52.84 mongod
```

Figure 3(a)



```
start time of process 1: 2023-12-01 1
5:04:11
<string>:1: RuntimeWarning: divide by
 zero encountered in divide
0 .. 10 .. 20 .. 30 .. 40 .. 50 .. 60
 .. 70 .. 80 .. 90 .. 100 - Done

real    1m17.515s
user    1m10.358s
sys     0m2.798s
end time of process 1: 2023-12-01 15:
05:29
```

Figure 3(b)

Figure 3. Single GDAL process statistics in monolithic environment. a) CPU utilization b) Turn-around Time

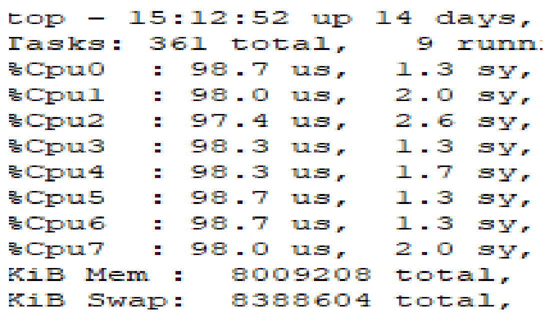
With the aforementioned observation it is understood that

ISSN (Online) 2394-2320

International Journal of Engineering Research in Computer Science and Engineering (IJERCSE)

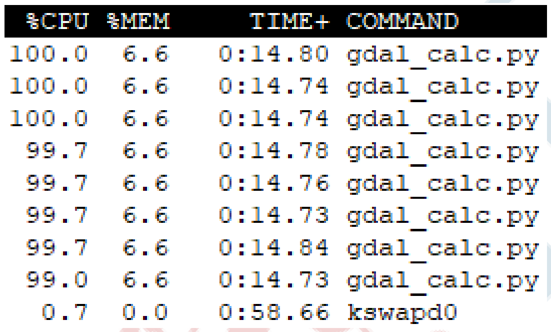
Volume 11 Issue 01 January 2024

the turn-around time of a process increases as the number of processes increase because the resources are limited and shared uniformly within the server. In IMGEOS, various levels of data processing systems say DIS, ADP, DPSG, etc. are allocated a set of servers in which, at times, resources of few systems are consumed to their threshold while peer system remain idle. Thus the mission critical processes are not able to utilize resources spread across the systems, this directly affects their performance by increasing the TAT of different processes.



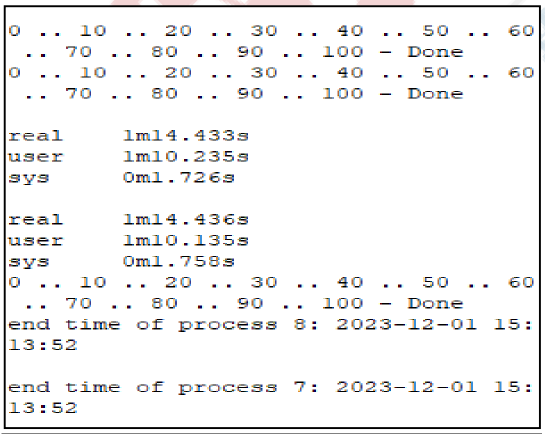
```
top - 15:12:52 up 14 days,
Tasks: 361 total,    9 runn
%Cpu0  : 98.7 us,  1.3 sy,
%Cpu1  : 98.0 us,  2.0 sy,
%Cpu2  : 97.4 us,  2.6 sy,
%Cpu3  : 98.3 us,  1.3 sy,
%Cpu4  : 98.3 us,  1.7 sy,
%Cpu5  : 98.7 us,  1.3 sy,
%Cpu6  : 98.7 us,  1.3 sy,
%Cpu7  : 98.0 us,  2.0 sy,
KiB Mem :  8009208 total,
KiB Swap:  8388604 total,
```

Figure 4(a)



```
 %CPU %MEM      TIME+ COMMAND
100.0  6.6    0:14.80 gdal_calc.py
100.0  6.6    0:14.74 gdal_calc.py
100.0  6.6    0:14.74 gdal_calc.py
 99.7  6.6    0:14.78 gdal_calc.py
 99.7  6.6    0:14.76 gdal_calc.py
 99.7  6.6    0:14.73 gdal_calc.py
 99.7  6.6    0:14.84 gdal_calc.py
 99.0  6.6    0:14.73 gdal_calc.py
  0.7  0.0    0:58.66 kswapd0
```

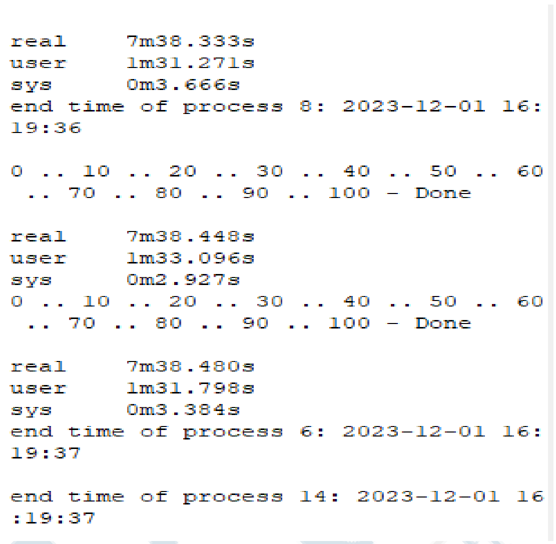
Figure 4(b)



```
0 .. 10 .. 20 .. 30 .. 40 .. 50 .. 60
 .. 70 .. 80 .. 90 .. 100 - Done
0 .. 10 .. 20 .. 30 .. 40 .. 50 .. 60
 .. 70 .. 80 .. 90 .. 100 - Done

real    1m14.433s
user    1m10.235s
sys     0m1.726s

real    1m14.436s
user    1m10.135s
sys     0m1.758s
0 .. 10 .. 20 .. 30 .. 40 .. 50 .. 60
 .. 70 .. 80 .. 90 .. 100 - Done
end time of process 8: 2023-12-01 15:
13:52

end time of process 7: 2023-12-01 15:
13:52
```

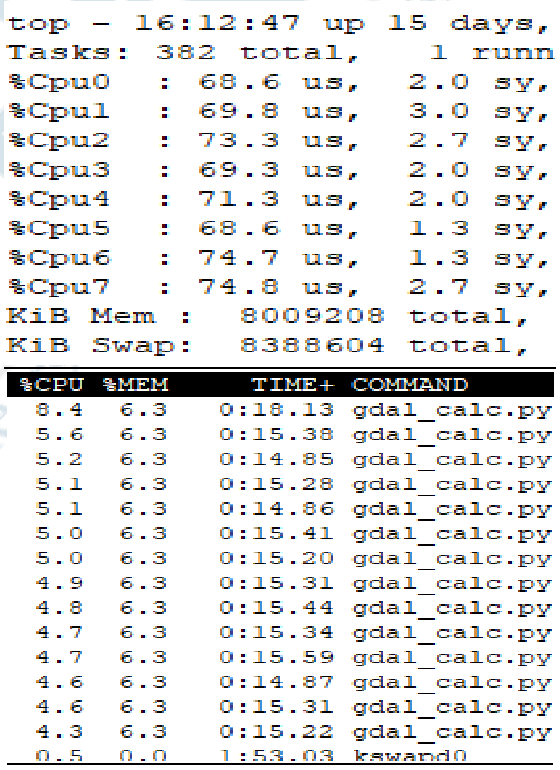
Figure 4(c)

Figure 4. Statistics of eight parallel GDAL process in monolithic environment. a) CPU utilization b) Turn-around Time



```
real    7m38.333s
user    1m31.271s
sys     0m3.666s
end time of process 8: 2023-12-01 16:
19:36

0 .. 10 .. 20 .. 30 .. 40 .. 50 .. 60
 .. 70 .. 80 .. 90 .. 100 - Done

real    7m38.448s
user    1m33.096s
sys     0m2.927s
0 .. 10 .. 20 .. 30 .. 40 .. 50 .. 60
 .. 70 .. 80 .. 90 .. 100 - Done

real    7m38.480s
user    1m31.798s
sys     0m3.384s
end time of process 6: 2023-12-01 16:
19:37

end time of process 14: 2023-12-01 16
:19:37
```

Figure 5(a)



```
top - 16:12:47 up 15 days,
Tasks: 382 total,   1 runn
%Cpu0  : 68.6 us,  2.0 sy,
%Cpu1  : 69.8 us,  3.0 sy,
%Cpu2  : 73.3 us,  2.7 sy,
%Cpu3  : 69.3 us,  2.0 sy,
%Cpu4  : 71.3 us,  2.0 sy,
%Cpu5  : 68.6 us,  1.3 sy,
%Cpu6  : 74.7 us,  1.3 sy,
%Cpu7  : 74.8 us,  2.7 sy,
KiB Mem :  8009208 total,
KiB Swap:  8388604 total,

 %CPU %MEM      TIME+ COMMAND
  8.4  6.3    0:18.13 gdal_calc.py
  5.6  6.3    0:15.38 gdal_calc.py
  5.2  6.3    0:14.85 gdal_calc.py
  5.1  6.3    0:15.28 gdal_calc.py
  5.1  6.3    0:14.86 gdal_calc.py
  5.0  6.3    0:15.41 gdal_calc.py
  5.0  6.3    0:15.20 gdal_calc.py
  4.9  6.3    0:15.31 gdal_calc.py
  4.8  6.3    0:15.44 gdal_calc.py
  4.7  6.3    0:15.34 gdal_calc.py
  4.7  6.3    0:15.59 gdal_calc.py
  4.6  6.3    0:14.87 gdal_calc.py
  4.6  6.3    0:15.31 gdal_calc.py
  4.3  6.3    0:15.22 gdal_calc.py
  0.5  0.0    1:53.03 kswapd0
```

Figure 5(b)

Figure 5. Statistics of fourteen parallel GDAL process in monolithic environment. a) CPU utilization b) Turn-around Time

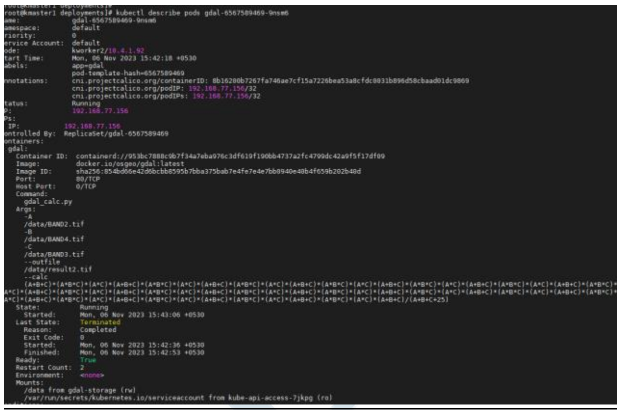
V. GDAL MODULE EXECUTION IN CONTAINER ORCHESTRATED PLATFORM.

In the test k8s architecture at IMGEOS, the GDAL software was installed as a container package on the two worker nodes which were configured with 8 CPU cores and 8GB RAM each. A deployment workload was built consisting container image of GDAL software and other networking and storage components required for the pods to run. The deployment was designed to perform arithmetic operations on CARTOSAT-2E TIFF files by running multiple GDAL pods across the cluster. In contrast to the native system, when a kubernetes workload deployment with 14 replica-sets of the GDAL container image was executed, 14 parallel pods were spun by kube-controller with 7 pods on each worker node as per resource available on the worker nodes. In this case the average Turn-around time for a pod is observed to be 17 sec (Figure 6b) with per pod resource utilization of 715 milli core CPU and 431MB RAM (Figure 6a).

Thus the resources available on two worker nodes can be utilized efficiently and multiple processes can consume resources spread across systems by consuming idle resources of peer system in the cluster. Whenever the pod completes its execution on a worker node, the resources allocated to it are released and are available for use by other pods on peer worker node. This leads to optimal resource utilization of systems in the cluster which ultimately helps in quicker execution of the container in the pods. Hence the performance is significantly improved as the turn-around time is reduced considerably.

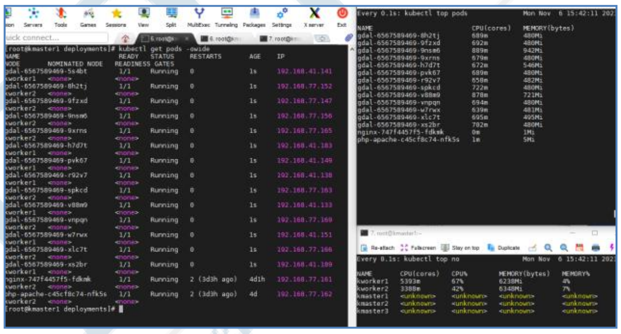
VI. FUTURE SCOPE & CONCLUSION

The future scope of work involves adding features like Log Management, Ingress Controller, Security Hardening, Centralized Repository etc. for proactive management of the kubernetes cluster. These features will enhance the capability of air-gapped IMGEOS kubernetes cluster. The aim of the paper was achieved by illustrating the difference in working of monolithic architecture and containerized architecture at IMGEOS and discussing the advantage of containerized architecture on the basis of resource utilization as well as the turn-around time of processes. After thorough understanding of resource utilization of systems in monolithic architecture as well as in containerized environment, it was learned that in a monolithic environment when a system is overloaded with multiple processes, because of limited availability of resources (CPU and Memory) on the system, each process is allocated fewer CPU and memory which directly affects the turn-around time of the processes. Also, the system is not able to utilize the idle resources available on its peer. Thus it causes inefficient resource utilization across the systems. While using container orchestrated platform, statistics show an improved Turn-around Time of the processes as shared pool of systems (aka cluster) allow idle resources on peer systems to be consumed when process requests are overloaded on the cluster. This significantly enhances the resource utilization capability of the systems in cluster. containerized mircoservices in IMGEOS network with significantly improved resource utilizing architecture.



```
root@kmaster1 deployments]# kubectl describe pods gdal-6567589469-9nsm6
ame:             gdal-6567589469-9nsm6
amespace:        default
riority:         0
ervice Account:  default
ode:             kworker2/10.4.1.92
tart Time:       Mon, 06 Nov 2023 15:42:18 +0530
abels:           app=gdal
                 pod-template-hash=6567589469
nnotations:      cni.projectcalico.org/containerID: 8b16280b7267fa746ae7cf15a7226bea53a8cfdc8031b896d58cbaad01dc9869
                 cni.projectcalico.org/podIP: 192.168.77.156/32
                 cni.projectcalico.org/podIPs: 192.168.77.156/32
tatus:           Running
P:               192.168.77.156
Ps:
 IP:           192.168.77.156
ontrolled By:  ReplicaSet/gdal-6567589469
ontainers:
 gdal:
   Container ID:  containerd://953bc7888c9b7f34a7eba976c3df619f190bb4737a2fc4799dc42a9f5f17df09
   Image:         docker.io/osgeo/gdal:latest
   Image ID:      sha256:854bd06e42d6bcbb8595b7bba375bab7e4fe7e4e7bb8940e40b4f659b202b40d
   Port:          80/TCP
   Host Port:     0/TCP
   Command:
     gdal_calc.py
   Args:
     -A
     /data/BAND2.tif
     -B
     /data/BAND4.tif
     -C
     /data/BAND3.tif
     --outfile
     /data/result2.tif
     --calc
     (A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*
A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*
A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)*(A*B*C)*(A*C)*(A+B+C)/(A+B+C+25)
   State:          Running
     Started:      Mon, 06 Nov 2023 15:43:06 +0530
   Last State:     Terminated
     Reason:       Completed
     Exit Code:    0
     Started:      Mon, 06 Nov 2023 15:42:36 +0530
     Finished:     Mon, 06 Nov 2023 15:42:53 +0530
   Ready:          True
   Restart Count:  2
   Environment:    <none>
   Mounts:
     /data from gdal-storage (rw)
     /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-7jkpg (ro)
```

Figure 6a



```
[root@kmaster1 deployments]# kubectl get pods -owide
NAME                          READY   STATUS    RESTARTS        AGE    IP
NODE       NOMINATED NODE   READINESS GATES
gdal-6567589469-5s4bt          1/1     Running   0               1s     192.168.41.141
kworker1   <none>           <none>
gdal-6567589469-8h2tj          1/1     Running   0               1s     192.168.77.152
kworker2   <none>           <none>
gdal-6567589469-9fzxd          1/1     Running   0               1s     192.168.77.147
kworker2   <none>           <none>
gdal-6567589469-9nsm6          1/1     Running   0               1s     192.168.77.156
kworker2   <none>           <none>
gdal-6567589469-9xrns          1/1     Running   0               1s     192.168.77.165
kworker2   <none>           <none>
gdal-6567589469-h7d7t          1/1     Running   0               1s     192.168.41.183
kworker1   <none>           <none>
gdal-6567589469-pvk67          1/1     Running   0               1s     192.168.41.149
kworker1   <none>           <none>
gdal-6567589469-r92v7          1/1     Running   0               1s     192.168.41.138
kworker1   <none>           <none>
gdal-6567589469-spkcd          1/1     Running   0               1s     192.168.77.163
kworker2   <none>           <none>
gdal-6567589469-v88m9          1/1     Running   0               1s     192.168.41.133
kworker1   <none>           <none>
gdal-6567589469-vnpqn          1/1     Running   0               1s     192.168.77.169
kworker2   <none>           <none>
gdal-6567589469-w7rwx          1/1     Running   0               1s     192.168.41.151
kworker1   <none>           <none>
gdal-6567589469-xlc7t          1/1     Running   0               1s     192.168.77.166
kworker2   <none>           <none>
gdal-6567589469-xs2br          1/1     Running   0               1s     192.168.41.189
kworker1   <none>           <none>
nginx-747f4457f5-fdkmk         1/1     Running   2 (3d3h ago)    4d1h   192.168.77.161
kworker2   <none>           <none>
php-apache-c45cf8c74-nfk5s     1/1     Running   2 (3d3h ago)    4d     192.168.77.162
kworker2   <none>           <none>
[root@kmaster1 deployments]#
```

```
Every 0.1s: kubectl top pods                  Mon Nov  6 15:42:11 202

NAME                          CPU(cores)   MEMORY(bytes)
gdal-6567589469-8h2tj         689m         480Mi
gdal-6567589469-9fzxd         692m         480Mi
gdal-6567589469-9nsm6         889m         942Mi
gdal-6567589469-9xrns         679m         480Mi
gdal-6567589469-h7d7t         672m         546Mi
gdal-6567589469-pvk67         689m         480Mi
gdal-6567589469-r92v7         658m         482Mi
gdal-6567589469-spkcd         722m         480Mi
gdal-6567589469-v88m9         878m         721Mi
gdal-6567589469-vnpqn         694m         480Mi
gdal-6567589469-w7rwx         639m         481Mi
gdal-6567589469-xlc7t         695m         495Mi
gdal-6567589469-xs2br         702m         480Mi
nginx-747f4457f5-fdkmk        0m           1Mi
php-apache-c45cf8c74-nfk5s    1m           5Mi
```

```
Every 0.1s: kubectl top no                    Mon Nov  6 15:42:11 202

NAME       CPU(cores)   CPU%        MEMORY(bytes)   MEMORY%
kworker1   5393m        67%         6238Mi          4%
kworker2   3388m        42%         6348Mi          7%
kmaster1   <unknown>    <unknown>   <unknown>       <unknown>
kmaster2   <unknown>    <unknown>   <unknown>       <unknown>
kmaster3   <unknown>    <unknown>   <unknown>       <unknown>
```

Figure 6b

Figure 6. Statistics of GDAL Container deployment executing 14 parallel pods in Kubernetes Environment. a) Resources utilization and deployment statistics of pods. b) Description of a pod stating the Turn-around Time of a pod.

REFERENCES

[1] Kubernetes Website: https://kubernetes.io/docs/concepts/containers/.

[2] Tigera Operator: https://docs.tigera.io/calico/latest/reference/installation/api#operator.tigera.io/v1.APIServr.

[3] Grzegorz Blinowski, Anna Ojdowska, Adam Przybyłek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation”, IEEE Access, Vol. 10, pp. 20357-20374, 2022.

[4] Kubernetes for Airgapped: https://kubernetes.io/blog/2023/10/12/bootstrap-an-air-gapped-cluster-with-kubea/