Extended ETL Framework on Cloud Resources and its engine “awsub”

Hiromu OCHIAI, Kenichi Chiba, Ai Okada and Yuichi Shiraishi

National Cancer Center Japan

Background

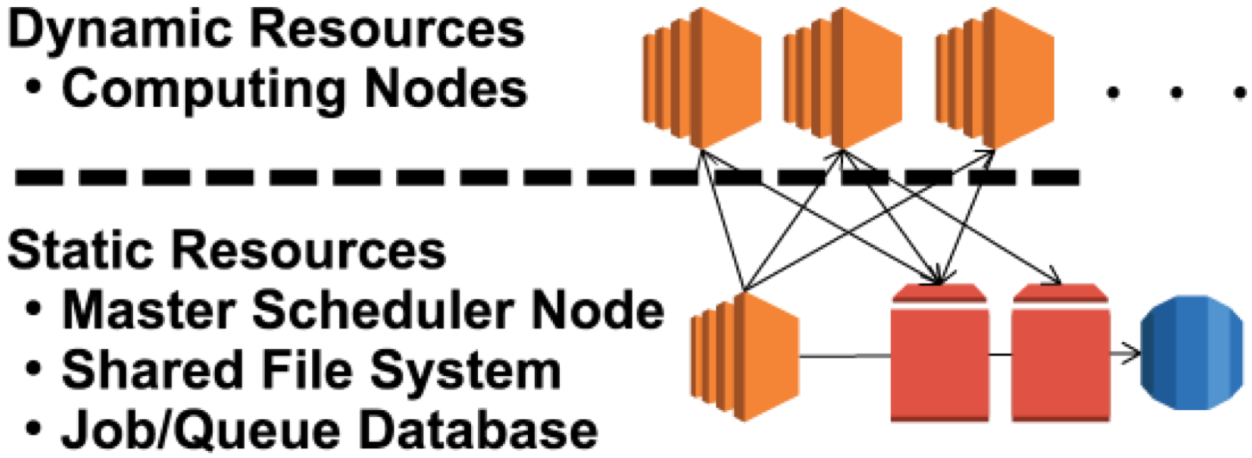
With the advancement of Cloud Services (which mean IaaS, PaaS, SaaS and so) researchers of biology and bioinformatics started to use cloud resources for genome analysis instead of using on-premise server resources.

Because most bioinformaticians are used to use huge clustered machines, building cluster is one of the way to execute genome analysis on cloud.

Here we suggest using ETL by which cloud resources can be procured more efficiently, and **“Extended ETL”** which solves potential problems of using plain ETL for genome analysis. And **“awsub”,** a command line tool, is working implementation of “Extended ETL” you can try.

Building Cluster on Cloud

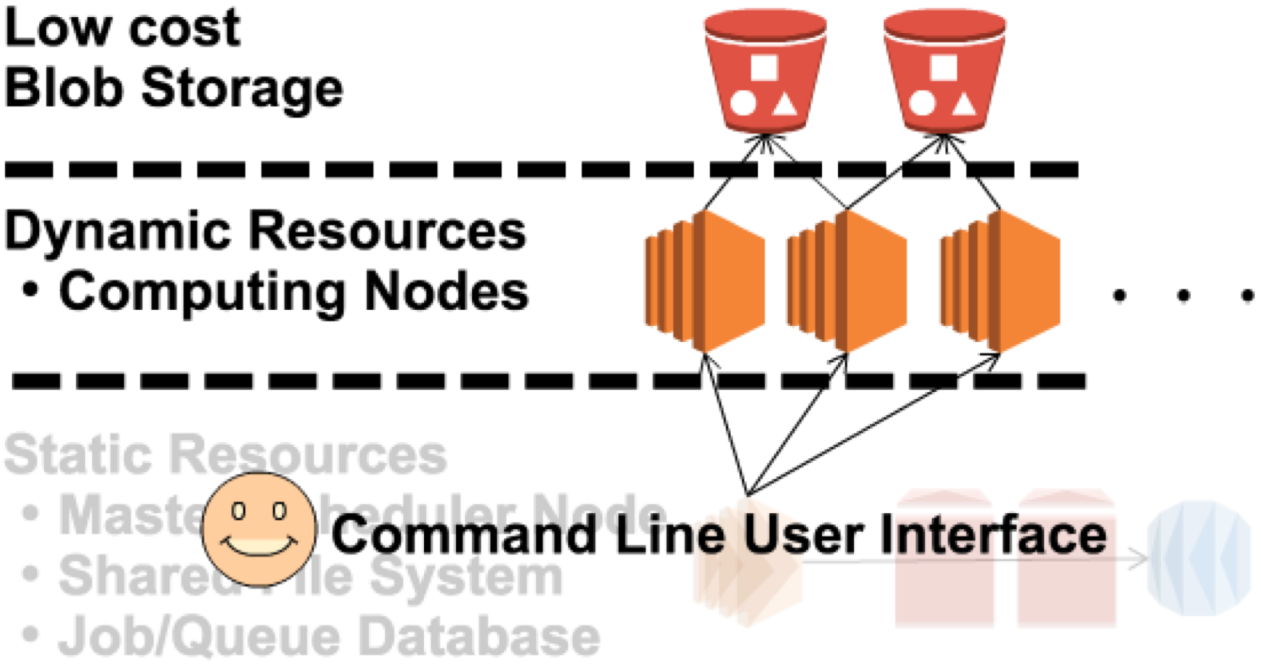
There are some softwares (Galaxy, ElastiCluster, CFN-Cluster, etc…) supporting building cluster on cloud resources. Submitting jobs to scheduler is what most bioinformaticians are used to. But it must have static resources in it.



This is an inefficiency of using cloud services, because “Cloud Service” is already-made cluster!

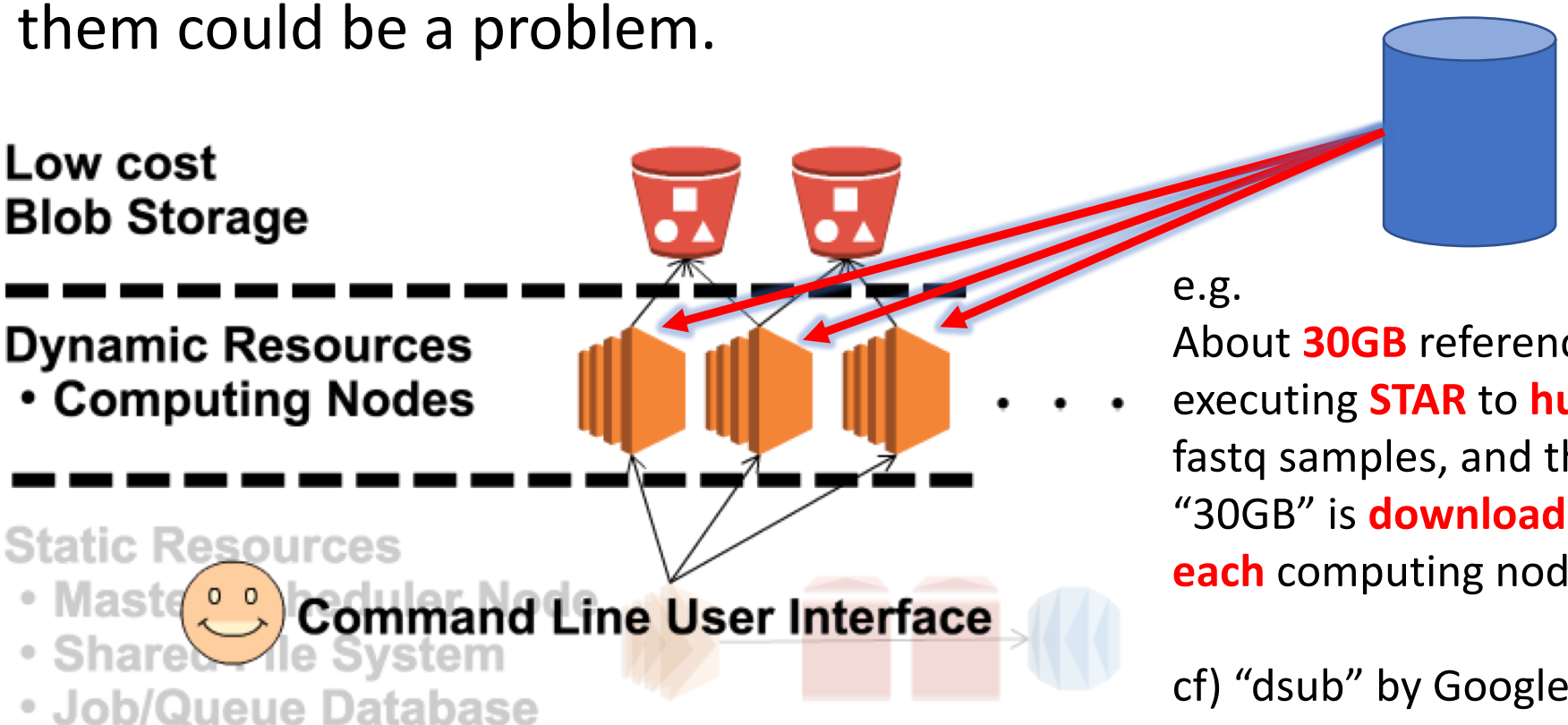
On-demand ETL model

"ETL" (Extract, Transform, Load) is a data processing model for general purpose. Combining with dynamic resource procurement, "on-demand ETL" works well for genome analysis, without any idling resources on cloud.



Huge Common Data matters

With using “on-demand ETL” on Cloud, it often matters that common data, reference genome for example, is so huge that network costs and instance time to download them could be a problem.

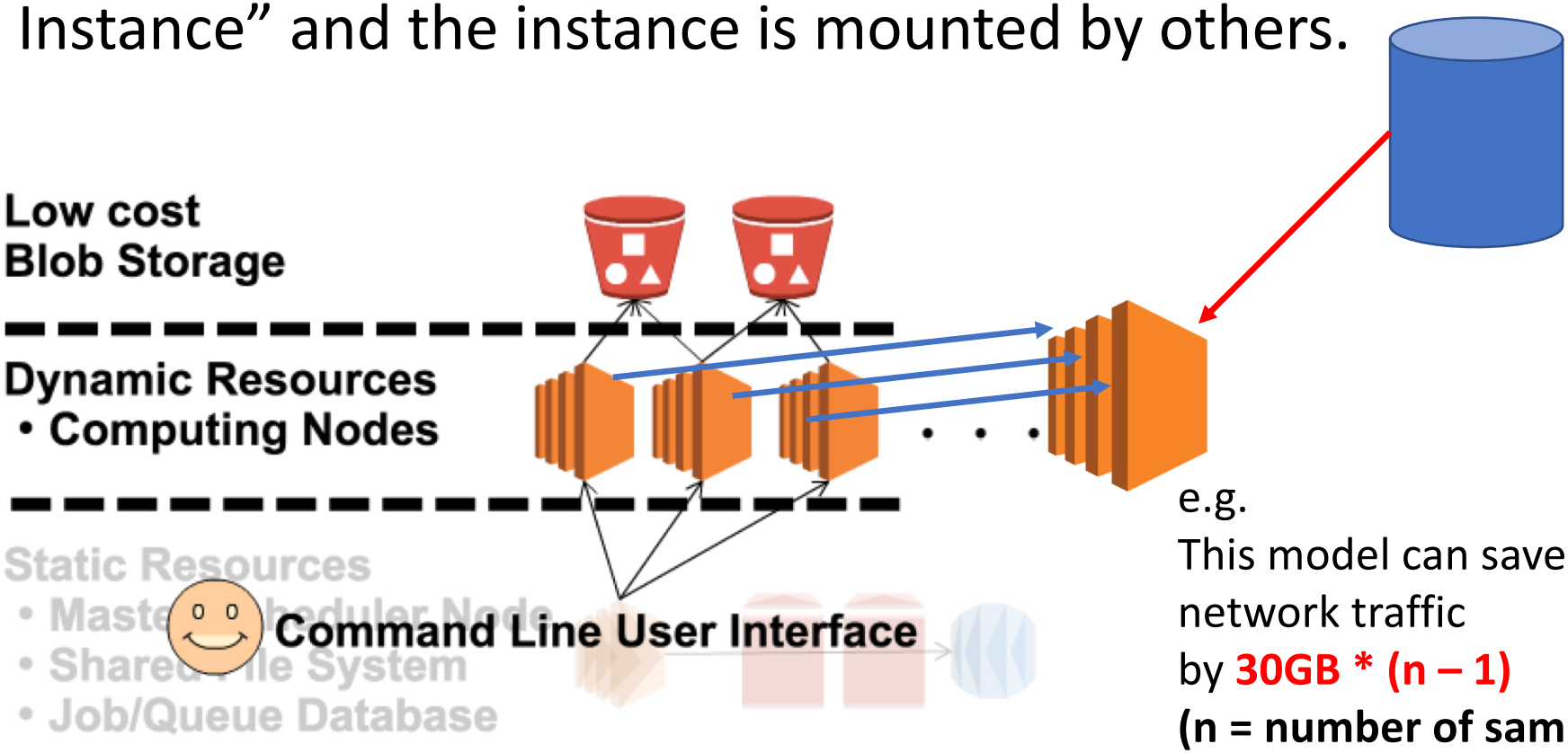


e.g.
About **30GB** reference for executing **STAR** to **human** fastq samples, and that “30GB” is **downloaded to each** computing node!

cf) “dsub” by Google Genomics

“ExTL”: Extended ETL model

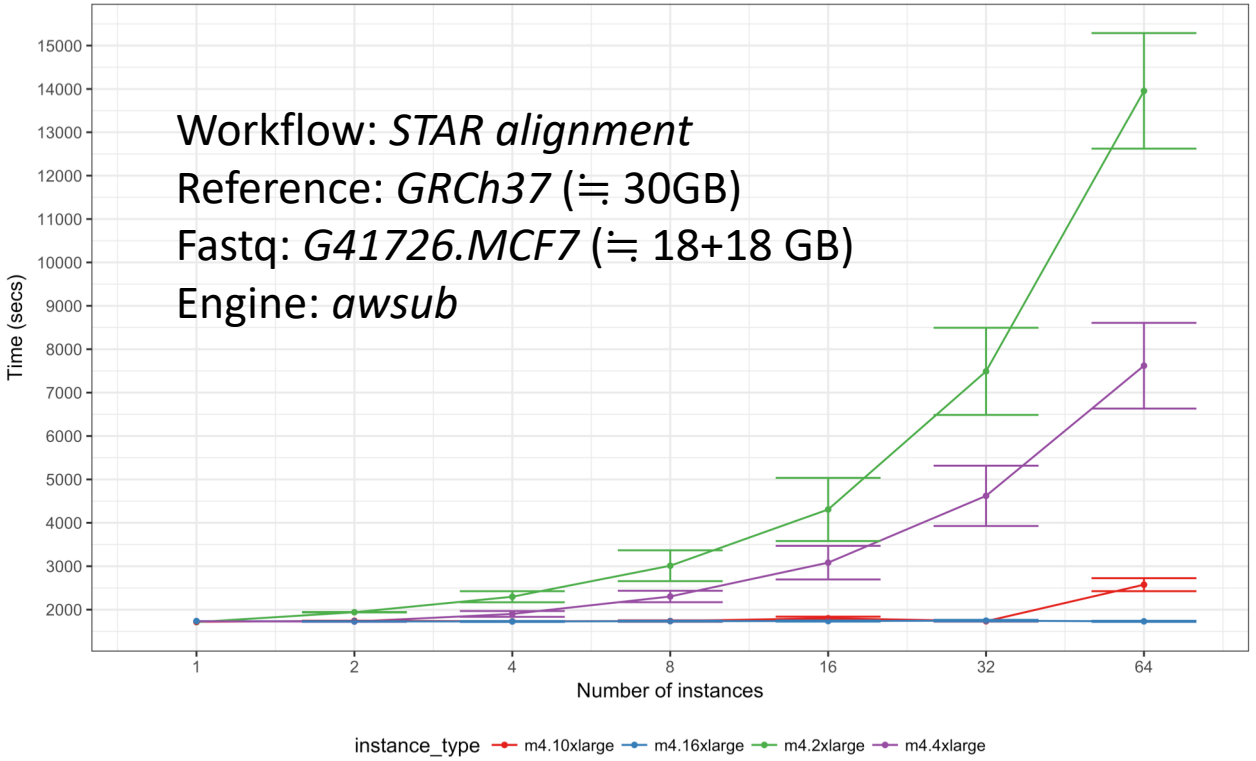
We suggest a novel framework with advantage to plain ETL model, called “ExTL”: Extended ETL model, in which huge common data are downloaded only once to “Shared Data Instance” and the instance is mounted by others.



e.g.
This model can save network traffic by **30GB * (n – 1)**
(n = number of samples)

Performance evaluation of ExTL

ExTL bundles duplicated downloading and the “Shared Data Instance (SDI)” is to be mounted by multiple computing nodes (with NFS in our implementation).

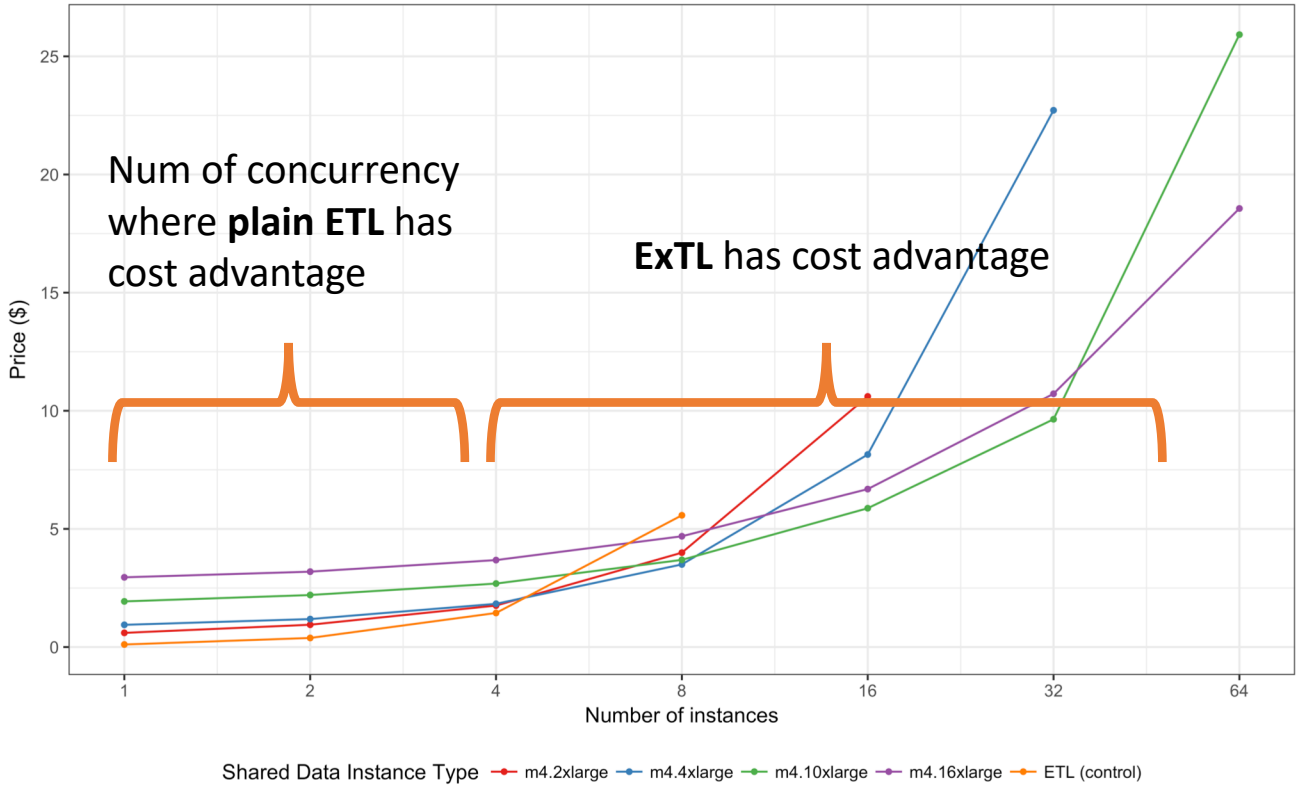


This figure describes how long STAR-alignment (RNA-seq) takes time by number of concurrency for fastq samples.

When using poor instance type for SDI, the execution time spikes heavily according to the concurrency.

Price evaluation of ExTL (logical)

Based on measured time and instance types actually used, the logical price can be evaluated for this example workflow.



Definitions

variable name	meaning	unit	note
D	Price of downloading data from storage	$/GB	
s	Size of reference data	GB	
t1(n)	Time for computing	sec	measured value
t2(s)	Time for downloading reference data	sec	measured value
n	Concurrency number	-	variable
P	Price of using computing instances	$/sec	
p	Price of block storage for reference data	$/sec	
Q	Price of using shared data instance	$/sec	

ETL

```
ETL ($) = (D * s + (P+P') * (t1+t2)) * n
```

ExTL

```
ExTL ($) = (D * s + (Q+P') * (t1+t2)) + (P * t1) * n
```

All the calculation logic are described on https://github.com/awsub/lab

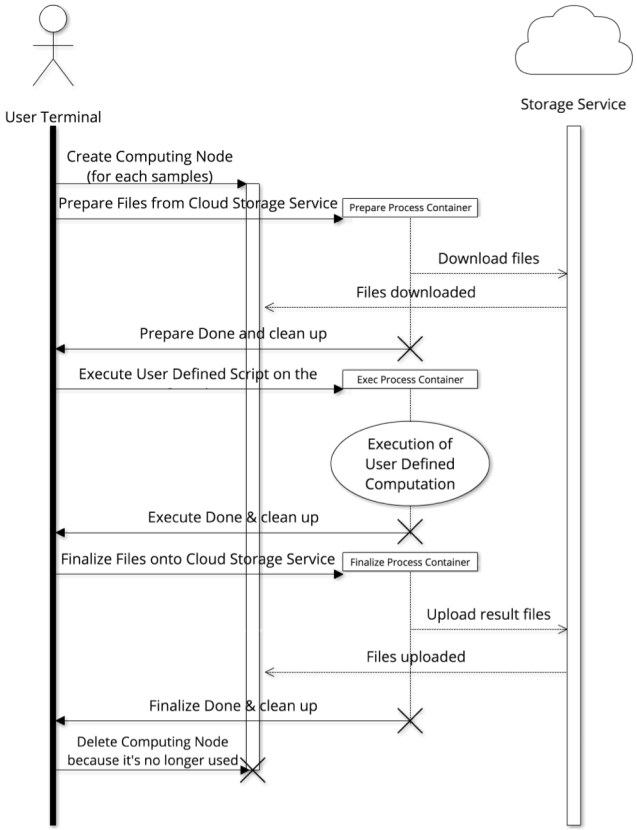
Check it out and star ;)

All is Implemented by “awsub”

We implemented a command line tool “awsub”, which is an engine for both ETL and ExTL on multiple* cloud services. (* ETL on AWS and GCP, ExTL on AWS so far)

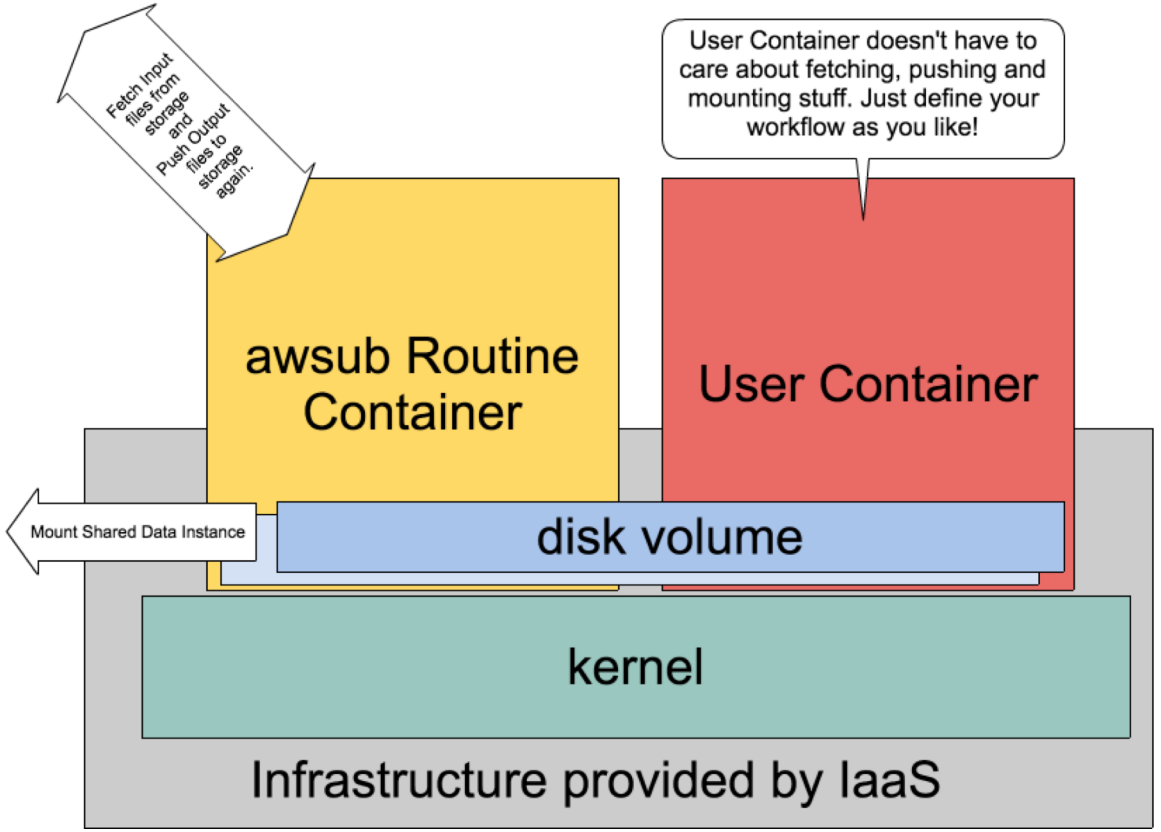
https://github.com/otiai10/awsub

```
$ awsub \
    --script ./my-workflow.sh \
    --image your/docker-image \
    --tasks ./input-files-location-list.csv \
    --aws-instance-type m4.16xlarge \ # if you want ;)
    --verbose
```



Container Architecture

By using “awsub”, user can execute any script on any runtime by any number of concurrency. This is implemented by container technology powered by Docker.

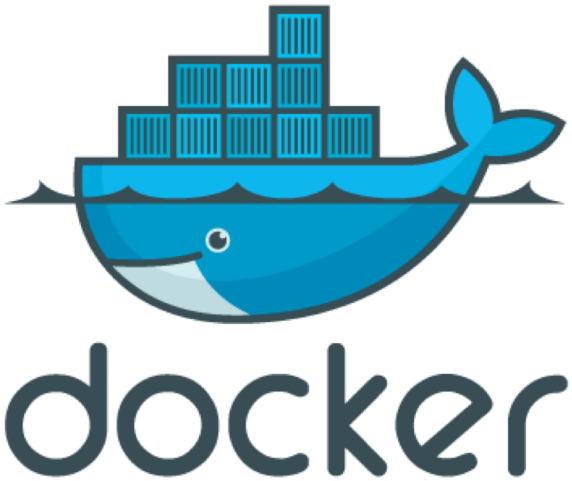



```go
// Construct creates containers inside job instance.
func (job *Job) Construct(shared *SharedData) (err error) {

	job.Lifetime(CONSTRUCT, "Constructing containers for this job...")

	volumes := []*daap.Volume{}
	if len(shared.Inputs) != 0 {
		volumes, err = shared.CreateNFSVolumesOn(job.Machine.Instance)
		if err != nil {
			return err
		}
		job.addContainerEnv(shared.Envs()...)
	}

	eg := new(errgroup.Group)
	eg.Go(func() error { return job.wakeupRoutineContainer() })
	eg.Go(func() error { return job.wakeupWorkflowContainer(volumes) })

	return eg.Wait()
}
```

https://github.com/otiai10/awsub/blob/master/core/job_construct.go

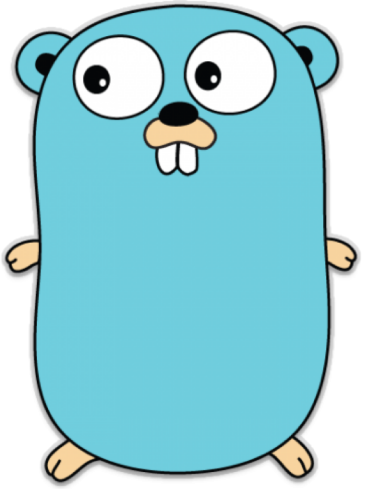
Application Architecture

Thanks for ***Docker*** to be implemented by ***Go***, “awsub” is using ***Docker*** source code directly as library SDK.



https://github.com/otiai10/daap

Bridge library implemented by Go



More advantage of using ***Go*** for “awsub”

- Static types
- Parallelization by goroutine
- Cross compiling binary for multiple platforms

Any questions and feature requests are welcome!!

Feel free to post any questions / issues on https://github.com/otiai10/awsub/issues , or just put questions here!

CWL support is coming soon!