The Cloud Harvester Architecture

The current Harvester, a python application, it is automatically launched by a batch script on a scheduled job every 5 minutes, and will harvest new data from FGP catalog and pushed them to the open Canada catalog.

The Cloud harvester on other hand will do the same thing as a lambda function over amazon AWS cloud.

In order to achieve that we have to understand how lambda service works. We will provide you with the architecture design of the cloud harvester explaining each module each services required to make it work.

# LAMBDA FUNCTION EXPLAINED

The way Lambda works is to allow a user two logical space of 128 mb in the cloud to run your function or application

## The “/var/tast” file system

The /var/task is the place where reside the lambda function or unzipped package. Here the file permission is limited to read only to ensure the integrity of lambda space with is 128mb. If you upload a package into lambda this is where lambda deploys it.

## The “/tmp” file system

Lambda allows user to control file permission from /tmp location. Here you can modify create file systems as long as the lambda function is running. Because in lambda nothing is persistent. Everything lives as long as the function is running.

The tmp location is used by the harvester for the following reasons:

* The need to read and write the following files:
* harvested\_record\_errors.html
* harvested\_record\_errors.csv
* harvested\_records.jl
* harvested\_records.xml
* run.last
* The harvester also need to log information about its current execution to help identify issues about the execution process.
* Some python module required by the harvester come with binary dependency that need execution permission. That is the case with CKANAPI.

From the aforementioned reason, the harvester is packaged into two files, on resources file package and the source files.

cloud-harvester-pack.zip. (source files )

harvesterResources-pack.zip. (resources files for the harvester)

## HARVESTER ARCHITECTURE

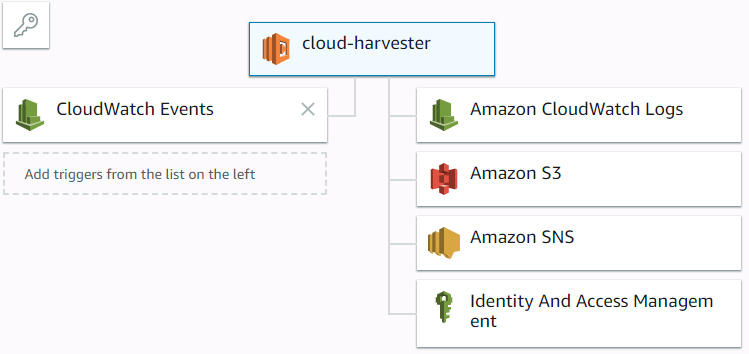


Fig.1.1: Harvester Design View

The harvester application in the AWS cloud is made of multiple components:

The cloud-harvester:

This is the deployment package itself where the lambda function and some python library are located. As a zip package, this component is uploaded through the lambda function interface.

## Amazon S3

Here we created the [fgp-s3-harvester](https://s3.console.aws.amazon.com/s3/) bucket to store [harvesterResources.zip](https://s3.amazonaws.com/fgp-s3-harvester/harvesterResources.zip) another component of the harvester within this package we have ckanapi binary and other resource files that will be deployed to the lambda tmp directory.

The harvester application will update this package at the end of every run to reuse some file. This is the only way to deal with persistence issue because in a lambda environment every thing is temporary.

## Amazon SNS

This component is attached to the harvester allowing the use of email notification service. We will use the notification service to send critical information to the support team about the harvester execution process. This service will be activated later.

## Harvester Operation Flow Diagram:

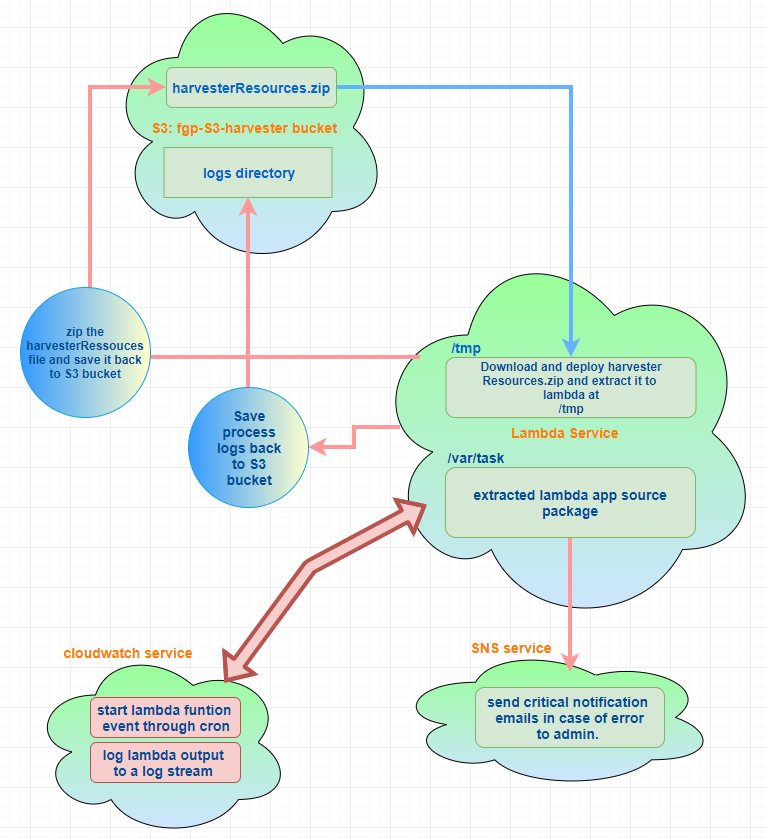


Fig:1.2: Flow diagram of the harvester operation in the cloud

## Lambda Function Execution Flow:

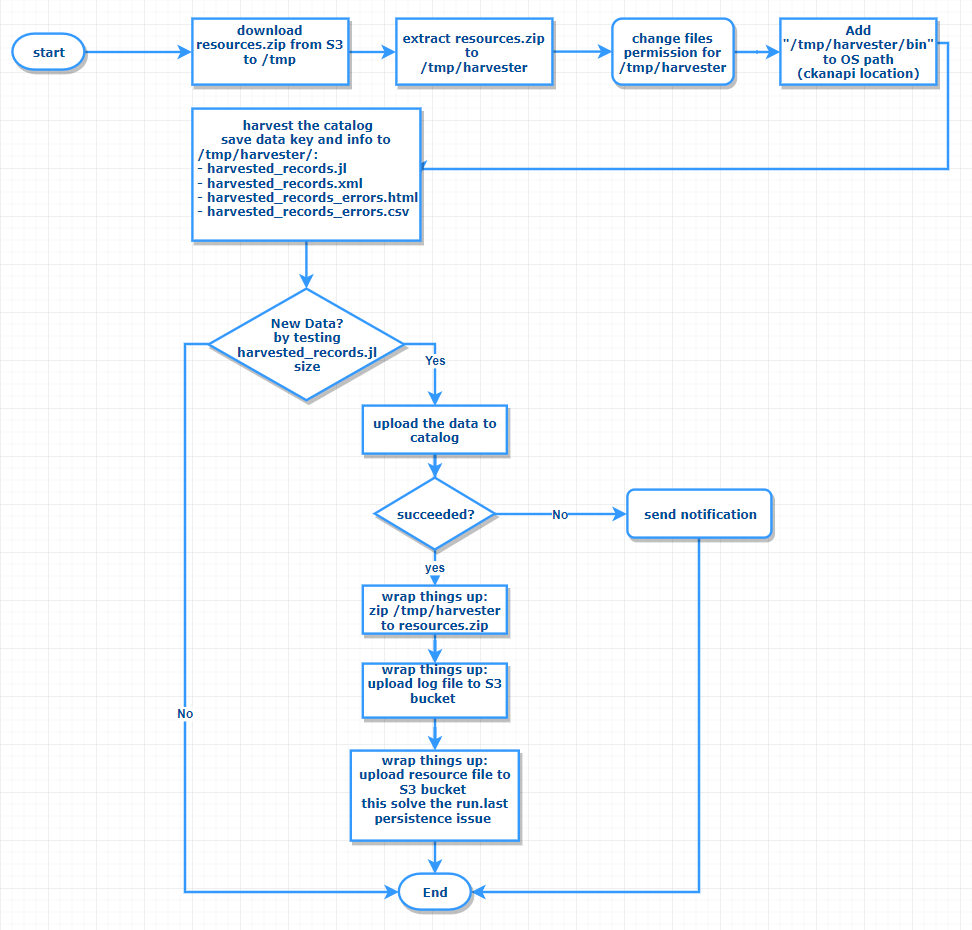


Fig.1.3: Execution flow diagram of the cloud-harvester

## Installation Procedure:

First, to make sure to run this application, the library dependencies need to be installed directly into the project directory and be part of the zipped package to be uploaded to AWS lambda.

From our dev environment we used the following command to install the libraries dependencies.

$>>pip install --upgrade -t <<library name>> /path/to/project/directory

Or this command from within the project directory:

$>>pip install --upgrade -t <<library name>> .

The installation come in two packages:

The cloud-harvester.zip and the harvesterResources.zip

In order to zip theses files, do not zip the entire folder but rather select all the file and folder from within and compress them as either clou-harvester.zip or harvesterResources.zip respectively.

### harvesterResources.zip structure:

The following shows the structure of the harvesterRessourcses.zip file

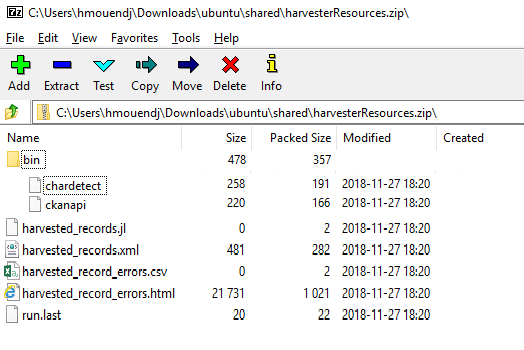


Fig:1.4: file structure of the harvesterResource.zip:

It is very important to keep this structure in order for the harvester to access the files with minimum configuration.

In the bin directory, we have a copy of ckanapi binary, as we need to change the permission of the file in order to execute it from within this folder. That is why this path is added to OS environment path.

### Cloud-Harvester zip: structure:

Keeping the structure of this zip file is important because when uploading the lambda package, we do not control the extraction into /var/task. We need to make sure that the lambda function handler is at that level.

The following Fig1.5 shows the structure of the lambda source package.

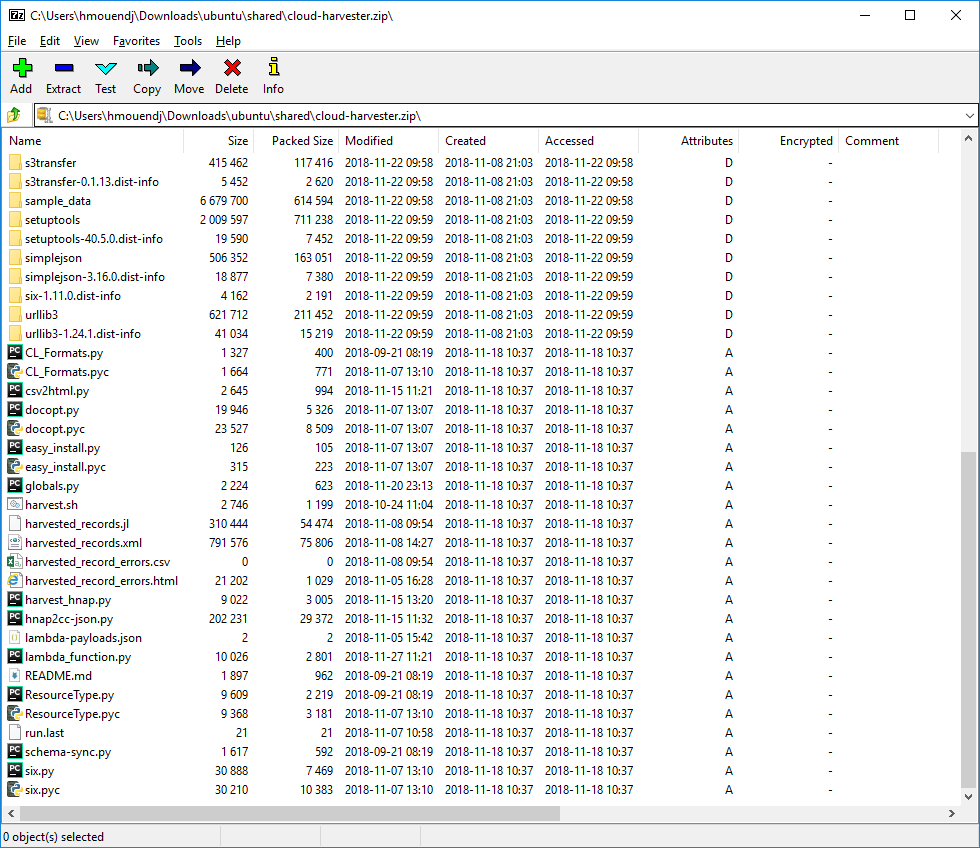


Fig.1.5: Source package for the harvester to be uploaded in Aws lambda

## Cloud Harvester Configuration:

We added some environment variables to the cloud-harvester dashboard in order to change the behaviour of this application in the fly without have touch the source code.

For instance, we can easily update the CKANAPI key or change the start time of the harvester from the lambda function dashboard and see the effect on the next harvester run.

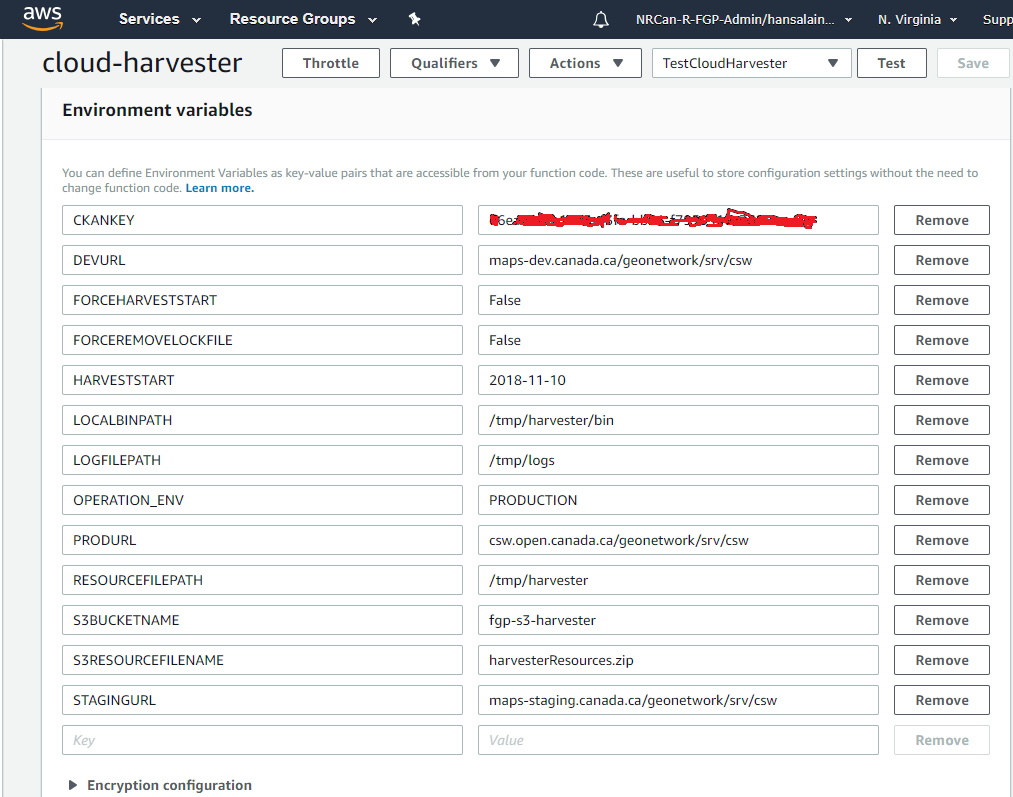


Fig.1.6: Environment variables for the cloud-harvester

## Environment Variables Definition:

CKANKEY**:** This variable holds the current ckanapi key use to access Open Canada

DEVURL**:** This variable holds the dev service url to upload dev data

FORCEHARVESTSTART: Set to True this will force the use of the time set in HARVESTSTART

HARVESTSTART: here the user can set the time the harvester will start harvesting from.

FORCEREMOVELOCKFILE: Set to true this will delete run.lock file.

LOCALBINPATH: the set the location of binary library path where to call binary dependencies (ckanapi for instance)

LOGFILEPATH: This is where the harvester application save log temporarily.

OPERATION\_ENV: This variable specify what is the harvester environment: Staging, Dev, Production

PRODURL: This variable holds the production service url to upload Production data

RESOURCEFILEPATH: specify where the resource files are extracted.

S3BUCKETNAME: Here we indicate the name of the S3 bucket to read and write to.

S3RESOURCEFILENAME: the name of the S3 bucket resource file to download and upload.

STAGINGURL: This variable holds the staging service url to upload staging data.

## Up coming development:

The harvester is currently deployed on the AWS provided sand box. Some issue or feature need to be addressed sooner or later:

* Ability to sending email notification in case off errors during harvesting process. this feature will be available in the future.
* Ability to clean the logs automatically keeping only small amount of logs
* To deploy this function to a staging and Dev environment.