scar Documentation

GRyCAP - I3M - UPV

May 17, 2023

Contents:

CHAPTER 1

About SCAR

SCAR is a framework to transparently execute containers out of Docker images in AWS Lambda, in order to run applications (see examples for ImageMagick, FFmpeg and AWS CLI, as well as deep learning frameworks such as Theano and Darknet) and code in virtually any programming language (see examples for Ruby, R, Erlang and Elixir) on AWS Lambda.

SCAR provides the benefits of AWS Lambda with the execution environment you decide, provided as a Docker image available in Docker Hub. It is probably the easiest, most convenient approach to run generic applications on AWS Lambda, as well as code in your favourite programming language, not only in those languages supported by AWS Lambda.

SCAR also supports a High Throughput Computing *Event-Driven File-Processing Programming Model* to create highly-parallel event-driven file-processing serverless applications that execute on customized runtime environments provided by Docker containers run on AWS Lambda. The development of SCAR has been published in the Future Generation Computer Systems scientific journal.

SCAR is integrated with API Gateway in order to expose an application via a highly-available HTTP-based REST API that supports both synchronous and asynchronous invocations. It is also integrated with AWS Batch. This way, AWS Lambda can be used to acommodate the execution of large bursts of short requests while long-running executions are delegated to AWS Batch.

SCAR allows to create serverless workflows by combining functions that run on either AWS Batch or AWS Lambda which produce output files that trigger the execution of functions that, again, run on either AWS Batch or AWS Lambda, using the very same Docker images, thus effectively creating highly-scalable cross-services serverless workflows.

SCAR has been developed by the Grid and High Performance Computing Group (GRyCAP) at the Instituto de Instrumentación para Imagen Molecular (I3M) from the Universitat Politècnica de València (UPV).

There is further information on the architecture of SCAR and use cases in the scientific publication "Serverless computing for container-based architectures" (pre-print available here), included in the Future Generation Computer Systems journal. Please acknowledge the use of SCAR by referencing the following cite:

```
A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, "Serverless computing for␣
↪container-based architectures," Futur. Gener. Comput. Syst., vol. 83, pp. 50-59,␣
↪Jun. 2018.
```

CHAPTER 2

Approach

SCAR provides a command-line interface to create a Lambda function to execute a container out of a Docker image stored in Docker Hub. Each invocation of the Lambda function will result in the execution of such a container (optionally executing a shell-script inside the container for further versatility).

The following underlying technologies are employed:

- udocker: A tool to execute Docker containers in user space.
  - The Fakechroot execution mode of udocker is employed, since Docker containers cannot be natively run on AWS Lambda. Isolation is provided by the boundary of the Lambda function itself.
- AWS Lambda: A serverless compute service that runs Lambda functions in response to events.

SCAR can optionally define a trigger so that the Lambda function is executed whenever a file is uploaded to an Amazon S3 bucket. This file is automatically made available to the underlying Docker container run on AWS Lambda so that an user-provided shell-script can process the file. See the *Event-Driven File-Processing Programming Model* for more details.

CHAPTER 3

Limitations

Unfortunately the AWS environment imposes several hard limits that are impossible to bypass:

- The Docker container must fit within the current AWS Lambda limits:
  - **Compressed + uncompressed** Docker image under **512 MB** (udocker needs to download the image before uncompressing it).
  - Maximum **execution time of 900 seconds** (15 minutes).
- Installation of packages in the user-defined script (i.e. using *yum*, *apt-get*, etc.) is currently not possible.

CHAPTER 4

Installation

If you want to avoid installing packages you can launch a docker container with scar installed. Please check the *SCAR container* section.

SCAR requires python3, pip3 and a configured AWS credentials file in your system. More info about the AWS credentials file can be found here.

You have to options when installing SCAR. You can use pip3 or you can clone the GitHub repository and install the required dependencies.

4.1 Install using pip3

1) Update setuptools to the latest version (or at least to version >= 40.8.0) with:

```
pip3 install -U setuptools
```

2) Install SCAR using the PyPI package with the command:

```
pip3 install scar
```

This will also creates an script in your local bin folder so you can execute the scar commands directly like: `scar ls`

4.2 Clone the Github Repository

1) Clone the GitHub repository:

```
git clone https://github.com/grycap/scar.git
cd scar
```

2) Install the Python required dependencies automatically with the command:

```
pip3 install -r requirements.txt
```

3) Execute the SCAR cli with the command:

```
python3 scar/scarcli.py ...
```

3) (Optional) Define an alias for easier usability:

```
alias scar='python3 `pwd`/scar/scarcli.py'
```

4.3 Extra dependencies

The last dependencies need to be installed using the package manager of your distribution (apt in this case):

```
sudo apt -y install zip unzip
```

CHAPTER 5

SCAR container

Other option to use SCAR is to create the container with the binaries included or to use the already available image with the packaged binaries installed from grycap/scar. Either you want to build the images from scratch or you want to use the already available image you will need Docker installed in your machine.

5.1 Building the SCAR image

All the steps needed to build the SCAR image are defined in the Dockerfile available at the root of the project. You only need to execute:

```
docker build -t scar -f Dockerfile .
```

This command creates a scar image in your docker repository that can be launched as:

```
docker run -it -v $AWS_CREDENTIALS_FOLDER:/home/scar/.aws -v $SCAR_CONFIG_FOLDER:/
↪home/scar/.scar scar
```

With the previous command we tell Docker to mount the folders required by SCAR (*~/.aws* and *~/.scar*) in the paths expected by the binary. Launching the container with the command described above also allow us to have different configuration folders wherever we want in our host machine.

Once we are inside the container you can execute SCAR like another system binary:

```
scar init -n scar-cowsay -i grycap/cowsay

scar run -n scar-cowsay

Request Id: 91e8afb6-8f19-11e8-9167-bd8a0b8b0f78
Log Group Name: /aws/lambda/scar-cowsay
Log Stream Name: 2018/07/24/[$LATEST]08444e77d6a14b09a47de0d5e4af5fa8
 ________________________________________
< Quick!! Act as if nothing has happened! >
 ----------------------------------------
```

(continues on next page)

(continued from previous page)

```
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

scar ls

NAME           MEMORY    TIME  IMAGE_ID       API_URL
-----------  --------  ------  -------------  ---------
scar-cowsay       512     300  grycap/cowsay  -

scar rm -n scar-cowsay
```

CHAPTER 6

Configuration

To use SCAR with AWS you need:

- Valid AWS IAM user credentials (Access Key and Secret Key ID) with permissions to deploy Lambda functions.
- An IAM Role for the Lambda function to be authorized to access other AWS services during its execution.

6.1 IAM User Credentials

The credentials have to be configured in your `$HOME/.aws/credentials` file (as when using AWS CLI). Check the AWS CLI documentation, specially section 'Configuration and Credential Files'.

6.2 IAM Role

The Lambda functions require an IAM Role in order to acquire the required permissions to access the different AWS services during its execution.

The following policy can be used in the IAM Role:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "logs:*"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    },
    {
      "Effect": "Allow",
      "Action": [
```

(continues on next page)

(continued from previous page)

```
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": "arn:aws:s3:::*"
    }
  ]
}
```

This IAM Role should be created beforehand. There is further documentation on this topic in the 'Creating IAM roles' section of the AWS documentation.

6.3 Configuration file

The first time you execute SCAR a default configuration file is created in the user location: `$HOME/.scar/scar.cfg`. As explained above, it is mandatory to set a value for the `aws.iam.role` property to use the Lambda service. If you also want to use the Batch service you have to update the values of the `aws.batch.compute_resources.security_group_ids`, and `aws.batch.compute_resources.subnets`. There is more information about the Batch usage here. Additionally, an explanation of all the configurable properties can be found in the example configuration file. Below is the complete default configuration file

```
{
  "scar": {
    "config_version": "1.0.9"
  },
  "aws": {
    "iam": {
      "boto_profile": "default",
      "role": ""
    },
    "lambda": {
      "boto_profile": "default",
      "region": "us-east-1",
      "execution_mode": "lambda",
      "timeout": 300,
      "memory": 512,
      "description": "Automatically generated lambda function",
      "runtime": "python3.7",
      "layers": [],
      "invocation_type": "RequestResponse",
      "asynchronous": false,
      "log_type": "Tail",
      "log_level": "INFO",
      "environment": {
        "Variables": {
          "UDOCKER_BIN": "/opt/udocker/bin/",
          "UDOCKER_LIB": "/opt/udocker/lib/",
          "UDOCKER_DIR": "/tmp/shared/udocker",
          "UDOCKER_EXEC": "/opt/udocker/udocker.py"
        }
      },
      "deployment": {
        "max_payload_size": 52428800,
        "max_s3_payload_size": 262144000
      },
```

(continues on next page)

(continued from previous page)

```
      "container": {
        "environment": {
          "Variables": {}
        },
        "timeout_threshold": 10
      },
      "supervisor": {
        "version": "1.2.0-rc4",
        "layer_name": "faas-supervisor",
        "license_info": "Apache 2.0"
      }
    },
    "s3": {
      "boto_profile": "default",
      "region": "us-east-1",
      "event": {
        "Records": [
          {
            "eventSource": "aws:s3",
            "s3": {
              "bucket": {
                "name": "{bucket_name}",
                "arn": "arn:aws:s3:::{bucket_name}"
              },
              "object": {
                "key": "{file_key}"
              }
            }
          }
        ]
      }
    },
    "api_gateway": {
      "boto_profile": "default",
      "region": "us-east-1",
      "endpoint": "https://{api_id}.execute-api.{api_region}.amazonaws.com/{stage_
↪name}/launch",
      "request_parameters": {
        "integration.request.header.X-Amz-Invocation-Type": "method.request.header.X-
↪Amz-Invocation-Type"
      },
      "http_method": "ANY",
      "method": {
        "authorizationType": "NONE",
        "requestParameters": {
          "method.request.header.X-Amz-Invocation-Type": false
        }
      },
      "integration": {
        "type": "AWS_PROXY",
        "integrationHttpMethod": "POST",
        "uri": "arn:aws:apigateway:{api_region}:lambda:path/2015-03-31/functions/
↪arn:aws:lambda:{lambda_region}:{account_id}:function:{function_name}/invocations",
        "requestParameters": {
          "integration.request.header.X-Amz-Invocation-Type": "method.request.header.
↪X-Amz-Invocation-Type"
        }
```

(continues on next page)

(continued from previous page)

```
      },
      "path_part": "{proxy+}",
      "stage_name": "scar",
      "service_id": "apigateway.amazonaws.com",
      "source_arn_testing": "arn:aws:execute-api:{api_region}:{account_id}:{api_id}/*
↪",
      "source_arn_invocation": "arn:aws:execute-api:{api_region}:{account_id}:{api_id}
↪/{stage_name}/ANY"
    },
    "cloudwatch": {
      "boto_profile": "default",
      "region": "us-east-1",
      "log_retention_policy_in_days": 30
    },
    "batch": {
      "boto_profile": "default",
      "region": "us-east-1",
      "vcpus": 1,
      "memory": 1024,
      "enable_gpu": false,
      "state": "ENABLED",
      "type": "MANAGED",
      "environment": {
        "Variables": {}
      },
      "compute_resources": {
        "security_group_ids": [],
        "type": "EC2",
        "desired_v_cpus": 0,
        "min_v_cpus": 0,
        "max_v_cpus": 2,
        "subnets": [],
        "instance_types": [
          "m3.medium"
        ],
        "launch_template_name": "faas-supervisor",
        "instance_role": "arn:aws:iam::{account_id}:instance-profile/ecsInstanceRole"
      },
      "service_role": "arn:aws:iam::{account_id}:role/service-role/AWSBatchServiceRole
↪"
    }
  }
}
```

CHAPTER 7

Basic Usage

7.1 Using a configuration file (recommended)

1) The most basic configuration file needed to launch a function is:

```
cat >> basic-cow.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-cowsay
      container:
        image: grycap/cowsay
EOF
```

   Where you define the name of the function and under it the image that will run inside the function.

2) Once you have created the file then you can create the function with:

```
scar init -f basic-cow.yaml
```

3) To execute the function the command is:

```
scar run -f basic-cow.yaml
```

4) Checking the function logs is as easy as:

```
scar log -f basic-cow.yaml
```

5) Finally to delete the function:

```
scar rm -f basic-cow.yaml
```

A role must be specified with the flag -r. It must be the ARN code of the role. To find it, search IAM, the Identity and Access Management of AWS. Press the Role tab. Search and select the role name. That will have an structure similar to this arn:aws:iam::{code}:role/{name_of_the_role}

7.2 Using CLI configuration (old school)

1) Create a Lambda function to execute a container (out of a Docker image that is stored in Docker Hub).

   In these examples the grycap/cowsay Docker image in Docker Hub will be employed:

```
scar init -n scar-cowsay -i grycap/cowsay
```

   Notice that the memory and time limits for the Lambda function can be specified in the command-line. Upon first execution, the file `$HOME/.scar/scar.cfg` is created with default values for the memory and timeout, among other features. The command-line values always take precedence over the values in the configuration file. The default values are 512 MB for the memory and 300 seconds for the timeout.

   Further information about the command-line arguments is available in the CLI help:

```
scar --help
```

2) Execute the Lambda function:

```
scar run -n scar-cowsay
```

   The first invocation to the Lambda function will pull the Docker image from DockerHub so it will take considerably longer than the subsequent invocations, which will most certainly reuse the existing cached Docker image.

3) Access the logs

   The logs are stored in CloudWatch with a default retention policy of 30 days (as default). The following command retrieves all the logs related to the Lambda function:

```
scar log -n scar-cowsay
```

   If you only want the logs related to a log-stream-name you can use:

```
scar log -n scar-cowsay -ls 'log-stream-name'
```

   And finally if you know the request id generated by your invocation, you can specify it to get the logs related:

```
scar log -n scar-cowsay -ri request-id
```

   You can also specify the log stream name to retrieve the values related with the request id, usually this will be faster if the function has generated a lot of log output:

```
scar log -n scar-cowsay -ls 'log-stream-name' -ri request-id
```

   All values are shown in the output when executing *scar log*. Do not forget to use the single quotes, as indicated in the example, to avoid unwanted shell expansions.

4) Remove the Lambda function

   You can remove the Lambda function together with the logs generated in CloudWatch by:

```
scar rm -n scar-cowsay
```

CHAPTER 8

Advanced Usage

8.1 Define a shell-script for each invocation of the Lambda function

Instead of packaging the script to be used inside the container image and having to modify the image each time you want to modify the script, you can specify a shell-script when initializing the Lambda function to trigger its execution inside the container on each invocation of the Lambda function. For example:

```
cat >> cow.sh << EOF
#!/bin/bash
/usr/games/cowsay "Executing init script !!"
EOF

cat >> cow.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-cowsay
      init_script: cow.sh
      container:
        image: grycap/cowsay
EOF

scar init -f cow.yaml
```

or using CLI parameters:

```
scar init -s cow.sh -n scar-cowsay -i grycap/cowsay
```

Now whenever this Lambda function is executed, the script will be run in the container:

```
scar run -f cow.yaml

Request Id: fb925bfa-bc65-47d5-beed-077f0de471e2
Log Group Name: /aws/lambda/scar-cowsay
```

(continues on next page)

(continued from previous page)

```
Log Stream Name: 2019/12/19/[$LATEST]0eb088e8a18d4599a572b7bf9f0ed321
 _____________________________
< Executing init script !! >
 -----------------------------
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

As explained in next section, this can be overridden by speciying a different shell-script when running the Lambda function.

8.2 Executing an user-defined shell-script

You can execute the Lambda function and specify a shell-script locally available in your machine to be executed within the container:

```
cat >> runcow.sh << EOF
#!/bin/bash
/usr/games/cowsay "Executing run script !!"
EOF

cat >> cow.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-cowsay
      run_script: runcow.sh
      container:
        image: grycap/cowsay
EOF

scar init -f cow.yaml
```

Now if you execute the function without passing more parameters, the entrypoint of the container is executed:

```
scar run -n scar-cowsay

Request Id: 97492a12-ca84-4539-be80-45696501ee4a
Log Group Name: /aws/lambda/scar-cowsay
Log Stream Name: 2019/12/19/[$LATEST]d5cc7a9db9b44e529873130f6d005fe1
 _____________________________________
/ No matter where I go, the place is \
\ always called "here".               /
 -------------------------------------
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

But, when you use the configuration file with the `run_script` property:

```
scar run -f cow.yaml
```

or use CLI parameters:

```
scar run -n scar-cowsay -s runcow.sh
```

or a combination of both (to avoid editing the initial .yaml file):

```
scar run -f cow.yaml -s runcow.sh
```

the passed script is executed:

```
Request Id: db3ff40e-ab51-4f90-95ad-7473751fb9c7
Log Group Name: /aws/lambda/scar-cowsay
Log Stream Name: 2019/12/19/[$LATEST]d5cc7a9db9b44e529873130f6d005fe1
 _____________________________
< Executing run script !! >
 -----------------------------
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

Have in mind that the script used in combination with the run command is no saved anywhere. It is uploaded and executed inside the container, but the container image is not updated. The shell-script needs to be specified and can be changed in each different execution of the Lambda function.

8.3 Passing environment variables

You can specify environment variables to the init command which will be in turn passed to the executed Docker container and made available to your shell-script. Using a configuration file:

```
cat >> cow.sh << EOF
#!/bin/bash
env | /usr/games/cowsay
EOF

cat >> cow-env.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-cowsay
      run_script: runcow.sh
      container:
        image: grycap/cowsay
        environment:
          Variables:
            TESTKEY1: val1
            TESTKEY2: val2
EOF

scar init -f cow-env.yaml
```

or using CLI parameters:

```
scar init -n scar-cowsay -i grycap/cowsay -e TEST1=45 -e TEST2=69 -s cow.sh
```

8.4 Executing custom commands and arguments

To run commands inside the docker image you can specify the command to be executed at the end of the command line. This command overrides any `init` or `run` script defined:

```
scar run -f cow.yaml df -h

Request Id: 39e6fc0d-6831-48d4-aa03-8614307cf8b7
Log Group Name: /aws/lambda/scar-cowsay
Log Stream Name: 2019/12/19/[$LATEST]9764af5bf6854244a1c9469d8cb84484
Filesystem      Size  Used Avail Use% Mounted on
/dev/root       526M  206M  309M  41% /
/dev/vdb        1.5G   21M  1.4G   2% /dev
```

8.5 Obtaining a JSON Output

For easier scripting, a JSON output can be obtained by including the *-j* or the *-v* (even more verbose output) flags:

```
scar run -f cow.yaml -j

{ "LambdaOutput":
  {
    "StatusCode": 200,
    "Payload": " _____\n/  \"I always avoid␣
↪prophesying beforehand \\\n| because it is much better               |\n|
↪                              |\n| to prophesy after the event has already |\n|␣
↪taken place. \" - Winston                |\n|
↪  |\n\\ Churchill                                /\n ---------------------------------
↪---------\n        \\    ^__^\n          \\  (oo)\\_______\n             (__)\\
↪)\\/\\\n                ||----w |\n                ||     ||\n",
    "LogGroupName": "/aws/lambda/scar-cowsay",
    "LogStreamName": "2019/12/19/[$LATEST]a4ba02914fd14ab4825d6c6635a1dfd6",
    "RequestId": "fcc4e24c-1fe3-4ca9-9f00-b15ec18c1676"
  }
}
```

8.6 Upload docker image files using an S3 bucket

SCAR allows to upload a saved docker image. We created the image file with the command `docker save grycap/cowsay > cowsay.tar.gz`. In case the docker is not in localhost pull it with the command `docker pull grycap/cowsay`:

```
cat >> cow.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-cowsay
```

(continues on next page)

(continued from previous page)

```
      container:
        image_file: cowsay.tar.gz
      deployment:
        bucket: scar-test
EOF

scar init -f cow.yaml
```

or for the CLI fans:

```
scar init -db scar-cowsay -n scar-cowsay -if cowsay.tar.gz
```

Have in mind that the maximum deployment package size allowed by AWS is an unzipped file of 250MB. The image file is unpacked in a temporal folder and the udocker layers are created. Depending on the size of the layers, SCAR will try to upload them or will show the user an error.

8.7 Upload 'slim' docker image files in the payload

Finally, if the image is small enough, SCAR allows to upload it in the function payload wich is ~50MB '' docker save grycap/minicow > minicow.tar.gz'' in case:

```
cat >> minicow.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-cowsay
      container:
        image_file: minicow.tar.gz
EOF

scar init -f minicow.yaml
```

To help with the creation of slim images, you can use minicon. Minicon is a general tool to analyze applications and executions of these applications to obtain a filesystem that contains all the dependencies that have been detected. By using minicon the size of the cowsay image was reduced from 170MB to 11MB.

8.8 Setting a specific VPC

You can also set an specific VPC parameters to configure the network in you lambda functions. You only have to add the `vpc` field setting the subnets and security groups as shown in the following example:

```
functions:
  aws:
  - lambda:
      vpc:
        SubnetIds:
          - subnet-00000000000000000
        SecurityGroupIds:
          - sg-00000000000000000
      name: scar-cowsay
      container:
        image: grycap/cowsay
```

CHAPTER 9

Using Lambda Image Environment

Scar uses by default the python3.7 Lambda environment using udocker program to execute the containers. In 2021 AWS added native support to ECR container images. Scar also supports to use this environment to execute your containers.

This functionality requires docker to be installed (check installation documentation here).

To use it you only have to set to `image` the lamda `runtime` property setting. You can set it in the scar configuration file:

```
{
  "aws": {
    "lambda": {
      "runtime": "image"
    }
  }
}
```

Or in the function definition file:

```
functions:
  aws:
  - lambda:
      runtime: image
      name: scar-function
      memory: 2048
      init_script: script.sh
      container:
        image: image/name
```

Or event set it as a parameter in the `init` scar call:

```
scar  init -f function_def.yaml -rt image
```

In this case the scar client will prepare the image and upload it to AWS ECR as required by the Lambda Image Environment.

To use this functionality you should use supervisor version 1.5.0 or newer.

Using the image runtime the scar client will build a new container image adding the supervisor and other needed files to the user provided image. This image will be then uploaded to an ECR registry to enable Lambda environment to create the function. So the user that executes the scar client must have the ability to execute the docker commands (be part of the `docker` group, see docker documentation)

9.1 Use alpine based images

Using the container image environment there is no limitation to use alpine based images (musl based). You only have to add the `alpine` flag in the function definition:

```
functions:
  aws:
  - lambda:
      runtime: image
      name: scar-function
      memory: 2048
      init_script: script.sh
      container:
        image: image/name
        alpine: true
```

If you use an alpine based image and you do not set the `alpine` flag you will get an execution Error:

```
Error: fork/exec /var/task/supervisor: no such file or directory
```

9.2 Use already prepared ECR images

You can also use a previously prepared ECR image instead of building it and and pushing to ECR. In this case you have to specify the full ECR image name and add set to false the `create_image` flag in the function definition:

```
functions:
  aws:
  - lambda:
      runtime: image
      name: scar-function
      memory: 2048
      init_script: script.sh
      container:
        image: 000000000000.dkr.ecr.us-east-1.amazonaws.com/scar-function
        create_image: false
```

But this ECR image must have been prepared to work with scar. So it must have the `init_script` and the `supervisor` installed and set it as the `CMD` of the docker image. You can use this example to create your own `Dockefile`:

```
from your_repo/your_image

# Create a base dir
ARG FUNCTION_DIR="/var/task"
WORKDIR ${FUNCTION_DIR}
# Set workdir in the path
```

(continues on next page)

(continued from previous page)

```
ENV PATH="${FUNCTION_DIR}:${PATH}"
# Add PYTHONIOENCODING to avoid UnicodeEncodeError as sugested in:
# https://github.com/aws/aws-lambda-python-runtime-interface-client/issues/19
ENV PYTHONIOENCODING="utf8"

# Copy your script, similar to:
# https://github.com/grycap/scar/blob/master/examples/darknet/yolo.sh
COPY script.sh ${FUNCTION_DIR}
# Download the supervisor binary
# https://github.com/grycap/faas-supervisor/releases/latest
# Copy the supervisor
COPY supervisor ${FUNCTION_DIR}
# Set it as the CMD
CMD [ "supervisor" ]
```

9.3 Do not delete ECR image on function deletion

By default the scar client deletes the ECR image in the function deletion process. If you want to maintain it for future functions you can modify the scar configuration file and set to false `delete_image` flag in the ecr configuration section:

```
{
  "aws": {
    "ecr": {
      "delete_image": false
    }
  }
}
```

Or set it in the function definition:

```
functions:
  aws:
  - lambda:
      runtime: image
      name: scar-function
      memory: 2048
      init_script: script.sh
      container:
        image: image/name
    ecr:
      delete_image: false
```

9.4 ARM64 support

Using the container image environment you can also specify the architecture to execute your lambda function (x86_64 or arm64) setting the architectures field in the function definition. If not set the default architecture will be used (x86_64):

```
functions:
  aws:
```

(continues on next page)

(continued from previous page)

```
  - lambda:
      runtime: image
      architectures:
        - arm64
      name: scar-function
      memory: 2048
      init_script: script.sh
      container:
        image: image/name
```

9.5 EFS support

Using the container image environment you can also configure file system access for your Lambda function. First you have to set the VPC parameters to use the same subnet where the EFS is deployed. Also verify that the iam role set in the scar configuration has the correct permissions and the Security Groups is properly configured to enable access to NFS port (see Configuring file system access for Lambda functions). Then you have to add the `file_system` field setting the arns and mount paths of the file systems to mount as shown in the following example:

```
functions:
  aws:
  - lambda:
      runtime: image
      vpc:
        SubnetIds:
          - subnet-0000000000000000
        SecurityGroupIds:
          - sg-0000000000000000
      file_system:
        - Arn: arn:aws:elasticfilesystem:us-east-1:000000000000:access-point/fsap-
↪0000000000000000
          LocalMountPath: /mnt/efs
      name: scar-function
      memory: 2048
      init_script: script.sh
      container:
        image: image/name
```

CHAPTER 10

API Gateway Integration

10.1 Define an HTTP endpoint

SCAR allows to transparently integrate an HTTP endpoint with a Lambda function via API Gateway. To enable this functionality you only need to define an API name and SCAR will take care of the integration process (before using this feature make sure you have to correct rights set in your aws account).

The following configuration file creates a generic api endpoint that redirects the http petitions to your lambda function:

```
cat >> api-cow.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-api-cow
      container:
        image: grycap/cowsay
    api_gateway:
      name: api-cow
EOF

scar init -f api-cow.yaml
```

After the function is created you can check the API URL with the command:

```
scar ls
```

That shows the basic function properties:

```
NAME                MEMORY     TIME   IMAGE_ID             API_URL                                 ␣
↪                                       SUPERVISOR_VERSION
----------------  --------  ------  ------------------  ------------------------------
↪-------------------------------------  --------------------
scar-api-cow           512      300  grycap/cowsay       https://r20bwcmdf9.execute-
↪api.us-east-1.amazonaws.com/scar/launch  1.2.0
```

10.2 CURL Invocation

You can directly invoke the API Gateway endpoint with `curl` to obtain the output generated by the application:

```
curl -s https://r20bwcmdf9.execute-api.us-east-1.amazonaws.com/scar/launch | base64 --
↪decode


 ________________________________________
/ Hildebrant's Principle:                \
|                                        |
| If you don't know where you are going, |
\ any road will get you there.           /
 ----------------------------------------
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

This way, you can easily provide an HTTP-based endpoint to trigger the execution of an application.

10.3 GET Request

SCAR also allows you to make an HTTP request, for that you can use the command `invoke` like this:

```
scar invoke -f api-cow.yaml

Request Id: e8cba9ee-5a60-4ff2-9e52-475e5fceb165
Log Group Name: /aws/lambda/scar-api-cow
Log Stream Name: 2019/12/20/[$LATEST]8aa8bdecba0647edae61e2e45e99ff90
 ________________________________________
/ What if everything is an illusion and \
| nothing exists? In that case, I       |
| definitely overpaid for my carpet.    |
|                                       |
\ -- Woody Allen, "Without Feathers"    /
 ----------------------------------------
      \   ^__^
       \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

This command automatically creates a *GET* request and passes the petition to the API endpoint defined previously. Bear in mind that the timeout for the API Gateway requests is 29s. Therefore, if the function takes more time to respond, the API will return an error message. To launch asynchronous functions you only need to add the `-a` parameter to the call:

```
scar invoke -f api-cow.yaml -a

Function 'scar-api-cow' launched successfully.
```

When you invoke an asynchronous function through the API Gateway there is no way to know if the function finishes successfully until you check the function invocation logs.

10.4 POST Request

You can also pass files through the HTTP endpoint. For the next example we will pass an image to an image transformation system. The following files were user to define the service:

```
cat >> grayify-image.sh << EOF
#! /bin/sh
FILE_NAME=`basename $INPUT_FILE_PATH`
OUTPUT_FILE=$TMP_OUTPUT_DIR/$FILE_NAME
convert $INPUT_FILE_PATH -type Grayscale $OUTPUT_FILE
EOF

cat >> image-parser.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-imagemagick
      init_script: grayify-image.sh
      container:
        image: grycap/imagemagick
      output:
      - storage_provider: s3
        path: scar-imagemagick/output
    api_gateway:
      name: image-api
EOF

scar init -f image-parser.yaml
```

We are going to convert this image.

To launch the service through the api endpoint you can use the following command:

```
scar invoke -f image-parser.yaml -db homer.png
```

The file specified after the parameter `-db` is codified and passed as the POST body. The output generated will be stored in the output bucket specified in the configuration file. Take into account that the file limitations for request response and asynchronous requests are 6MB and 128KB respectively, as specified in the AWS Lambda documentation.

The last option available is to store the output wihtout bucket intervention. What we are going to do is pass the generated files to the output of the function and then store them in our machine. For that we need to slightly modify the script and the configuration file:

```
cat >> grayify-image.sh << EOF
#! /bin/sh
FILE_NAME=`basename $INPUT_FILE_PATH`
OUTPUT_FILE=$TMP_OUTPUT_DIR/$FILE_NAME
convert $INPUT_FILE_PATH -type Grayscale $OUTPUT_FILE
cat $OUTPUT_FILE
EOF

cat >> image-parser.yaml << EOF
functions:
  aws:
```

(continues on next page)

(continued from previous page)

```
  - lambda:
      name: scar-imagemagick
      init_script: grayify-image.sh
      container:
        image: grycap/imagemagick
    api_gateway:
      name: image-api
EOF

scar init -f image-parser.yaml
```

This can be achieved with the command:

```
scar invoke -f image-parser.yaml -db homer.png -o grey_homer.png
```

CHAPTER 11

Function Definition Language (SCAR)

Example:

```
functions:
  aws:
  - lambda:
      boto_profile: default
      region: us-east-1
      name: function1
      memory: 1024
      timeout: 300
      execution_mode: lambda
      log_level: INFO
      layers:
      - arn:....
      environment:
        Variables:
          KEY1: val1
          KEY2: val2
      init_script: ffmpeg-script.sh
      container:
        image: jrottenberg/ffmpeg:4.1-ubuntu
        timeout_threshold": 10
        environment:
          Variables:
            KEY1: val1
            KEY2: val2
      input:
      - storage_provider: minio.my_minio
        path: my-bucket/test
      output:
      - storage_provider: s3.my_s3
        path: my-bucket/test-output
        suffix:
          - wav
```

(continues on next page)

(continued from previous page)

```
        - srt
      prefix:
        - result-
    supervisor:
      version: latest


  iam:
    boto_profile: default
    role: ""
  api_gateway:
    boto_profile: default
    region: us-east-1
  cloudwatch:
    boto_profile: default
    region: us-east-1
    log_retention_policy_in_days: 30
  batch:
    boto_profile: default
    region: us-east-1
    vcpus: 1
    memory: 1024
    enable_gpu: False
    service_role: "arn:..."
    environment:
      Variables:
        KEY1: val1
        KEY2: val2
    compute_resources:
      security_group_ids:
      - sg-12345678
      desired_v_cpus: 0
      min_v_cpus: 0
      max_v_cpus: 2
      subnets:
      - subnet-12345
        subnet-67891
      instance_types:
      - "m3.medium"
      instance_role: "arn:..."
oscar:
- my_oscar:
    name: service1
    memory: 1Gi
    cpu: '1.0'
    log_level: INFO
    image: grycap/darknet
    script: my-script.sh
    environment:
      Variables:
        KEY1: val1
        KEY2: val2
    input:
    - storage_provider: minio.my_minio
      path: my-bucket/test
    output:
    - storage_provider: s3.my_s3
```

(continues on next page)

(continued from previous page)

```
        path: my-bucket/test-output
        suffix:
          - wav
          - srt
        prefix:
          - result-
storage_providers:
  s3:
    my_s3:
      access_key: awsuser
      secret_key: awskey
      region: us-east-1
  minio:
    my_minio:
      endpoint: minio-endpoint
      verify: True
      region: us-east-1
      access_key: muser
      secret_key: mpass
  onedata:
    my_onedata:
      oneprovider_host: op-host
      token: mytoken
      space: onedata_space
```

11.1 Top level parameters

Field	Description
functions *Functions*.	Map to define the credentials for a MinIO storage provider, being the key the user-defined identifier for the provider
storage_providers OSCAR-FDL-StorageProvider	Parameter to define the credentials for the storage providers to be used in the services, in lambda function only s3 buckets are allowed.

11.2 Functions

Field	Description
AWS Elements *Lambda*	Parameters to define manage AWS resources
oscar OSCAR-FDL	Parameters to define OSCAR services

11.3 AWS Elements

Field	Description
Lambda *Lambda*	Set Lambda properties.
iam *IAM*	Set IAM properties.
api_gateway *API Gateway*	Set API Gateway properties.
cloudwatch *Cloudwatch*	Set CloudWatch properties.
batch *AWS Batch*	Set AWS Batch properties.

11.4 Lambda

Field	Description
boto_profile *string*	Boto profile used for the lambda client. Default 'default'.Must match the profiles in the file ~/.aws/credentials
region *string*	Region of the function, can be any region supported by AWS.
name *string*	Function's name. REQUIRED
memory *string*	Memory of the function, in MB, min 128, max 3008. Default '512'
timeout *string*	Maximum execution time in seconds, max 900. Default '300'
execution_mode *string*	Set job delegation or not. Possible values 'lambda', 'lambda-batch', 'batch'. Default 'lambda'.
log_level *string*	Supervisor log level. Can be INFO, DEBUG, ERROR, WARNING. Default 'INFO'
layers *string array*	Lambda function's layers arn (max 4). SCAR adds the supervisor layer automatically.
environment OSCAR-FDL-Enviroment	Environment variables of the function. This variables are used in the lambda's environment, not the container's environment.
init_script *string*	Script executed inside of the function's container.
container *Container*	Define udocker container properties
input OSCAR-FDL-StorageIOConfig	Define input storage providers linked with the function.
output OSCAR-FDL-StorageIOConfig	Define input storage providers linked with the function.
supervisor *Supervisor*	Properties for the faas-supervisor used in the inside the lambda function

11.5 Container

Field	Description
image *string*	Container image to use. REQUIRED
timeout_threshold *string*	Time used to post-process data generated by the container. This time is substracted from the total time set for the function. If there are a lot of files to upload as output, maybe this value has to be increased. Default '10' seconds.
environment OSCAR-FDL-Enviroment	Environment variables of the container. These variables are passed to the container environment, that is, can be accessed from the user's script.

11.6 Supervisor

Field	Description
version *string*	Must be a Github tag or "latest". Default 'latest'.

11.7 IAM

Field	Description
boto_profile *string*	Boto profile used for the iam client.
role *string*	The Amazon Resource Name (ARN) of the function's execution role. This value is usually set for all the functions in the SCAR's default configuration file. REQUIRED

11.8 API Gateway

Field	Description
boto_profile *string*	Boto profile used for the iam client.
region *string*	Region of the function, can be any region supported by AWS.

11.9 Cloudwatch

Field	Description
boto_profile *string*	Boto profile used for the iam client.
region *string*	Region of the function, can be any region supported by AWS.
log_retention_policy_in_days *string*	Number of days that the functions logs are stored.

11.10 AWS Batch

Field	Description
boto_profile *string*	Boto profile used for the iam client.
region *string*	Region of the function, can be any region supported by AWS.
vcpus *string*	The number of vCPUs reserved for the container. Used in the job definition. Default 1
memory *string*	The hard limit (in MiB) of memory to present to the container. Used in the job definition. Default 1024
enable_gpu *string*	Request GPU resources for the launched container. Default 'False'. Values 'False', 'True'
service_role *string*	Environment variables passed to the batch container
environment OSCAR-FDL-Enviroment	Number of days that the functions logs are stored.
compute_resources *Compute Resources*	Parameters that specifies all the resources is going to use.

11.11 Compute Resources

Field	Description
security_group_ids *string array*	List of the Amazon EC2 security groups associated with instances launched in the compute environment. REQUIRED when using batch.
desired_vcpus *string*	The desired number of Amazon EC2 vCPUS in the compute environment. Default 0
min_vcpus *string*	The minimum number of Amazon EC2 vCPUs that an environment should maintain. Default 0
max_vcpus *string*	The maximum number of Amazon EC2 vCPUs that an environment should maintain. Default 2
subnets *string array*	List of the VPC subnets into which the compute resources are launched. REQUIRED when using batch
instance_types *string array*	The instances types that may be launched. You can specify instance families to launch any instance type within those families. (for example, c5 or p3), or you can specify specific sizes within a family (such as c5.8xlarge). You can also choose optimal to pick instance types (from the C, M, and R instance families) on the fly that match the demand of your job queues. Default 'm3.medium'
instance_role *string*	The Amazon ECS instance profile applied to Amazon EC2 instances in a compute environment.

CHAPTER 12

AWS Batch Integration

AWS Batch allows to efficiently execute batch computing jobs on AWS by dynamically provisioning the required underlying EC2 instances on which Docker-based jobs are executed. SCAR allows to transparently integrate the execution of the jobs through AWS Batch. Three execution modes are now available in SCAR:

- *lambda*: This is the default execution mode. All executions will be run on AWS Lambda.
- *lambda-batch*: Executions will be run on AWS Lambda. If the default timeout is reached, then the execution is automatically delegated to AWS Batch.
- *batch*: Executions will be automatically diverted to AWS Batch.

This way, you can use AWS Lambda as a highly-scalable cache for burts of short computational jobs while longer executions can be automatically delegated to AWS Batch. The very same programming model is maintained regardless of the service employed to perform the computation.

12.1 Set up your configuration file

To be able to use AWS Batch, first you need to set up your configuration file, located in *~/.scar/scar.cfg*

The variables responsible for batch configuration are:

```
"batch": {
  "boto_profile": "default",
  "region": "us-east-1",
  "vcpus": 1,
  "memory": 1024,
  "enable_gpu": false,
  "state": "ENABLED",
  "type": "MANAGED",
  "environment": {
    "Variables": {}
  },
  "compute_resources": {
```

(continues on next page)

(continued from previous page)

```
    "security_group_ids": [],
    "type": "EC2",
    "desired_v_cpus": 0,
    "min_v_cpus": 0,
    "max_v_cpus": 2,
    "subnets": [],
    "instance_types": [
      "m3.medium"
    ],
    "launch_template_name": "faas-supervisor",
    "instance_role": "arn:aws:iam::{account_id}:instance-profile/ecsInstanceRole"
  },
  "service_role": "arn:aws:iam::{account_id}:role/service-role/AWSBatchServiceRole"
}
```

Since AWS Batch deploys Amazon EC2 instances, the REQUIRED variables are:

- *security_group_ids*: The EC2 security group that is associated with the instances launched in the compute environment. This allows to define the inbound and outbound network rules in order to allow or disallow TCP/UDP traffic generated from (or received by) the EC2 instance. You can choose the default VPC security group.
- *subnets*: The VPC subnet(s) identifier(s) on which the EC2 instances will be deployed. This allows to use multiple Availability Zones for enhanced fault-tolerance.

The remaining variables have default values that should be enough to manage standard batch jobs. The default fdl file explains briefly the remaining Batch variables and how are they used.

Additional info about the variables and the different values that can be assigned can be found in the AWS API Documentation.

12.2 Set up your Batch IAM role

The default IAM role used in the creation of the EC2 for the Batch Compute Environment is **arn:aws:iam::$ACCOUNT_ID:instance-profile/**ecsInstanceRole****. Thus, if you want to provide S3 access to your Batch jobs you have to specify the corresponding policies in the aforementioned role. If you have a role aleredy configured, you can set it in the configuration file by changin the variable *batch.compute_resources.instance_role*.

12.3 Define a job to be executed in batch

To enable this functionality you only need to set the execution mode of the Lambda function to one of the two available used to create batch jobs ('lambda-batch' or 'batch') and SCAR will take care of the integration process (before using this feature make sure you have the correct rights set in your AWS account).

As an example, the following configuration file defines a Lambda function that creates an AWS Batch job to execute the plants classification example (all the required scripts and example files used in this example can be found there):

```
cat >> scar-plants.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-plants
      init_script: bootstrap-plants.sh
```

(continues on next page)

(continued from previous page)

```
      memory: 1024
      execution_mode: batch
      container:
        image: deephdc/deep-oc-plant-classification-theano
      input:
      - storage_provider: s3
        path: scar-plants/input
      output:
      - storage_provider: s3
        path: scar-plants/output
EOF
```

You can then create the function:

```
scar init -f scar-plants.yaml
```

Additionally for this example to run you have to upload the execution script to S3:

```
scar put -b scar-plants -p plant-classification-run.sh
```

Once uploaded you have to manually set their access to public so it can be accessed from batch. This has to be done to deal with the batch limits as it is explained in the next section.

And trigger the execution of the function by uploading a file to be processed to the corresponding folder:

```
scar put -b scar-plants/input -p daisy.jpg
```

SCAR automatically creates the compute environment in AWS Batch and submits a job to be executed. Input and output data files are transparently managed as well according to the programming model.

The CloudWatch logs will reveal the execution of the Lambda function as well as the execution of the AWS Batch job. Notice that whenever the execution of the AWS Batch job has finished, the EC2 instances will be eventually terminated. Also, the number of EC2 instances will increase and shrink to handle the incoming number of jobs.

12.4 Combine AWS Lambda and AWS Batch executions

As explained in the section *Event-Driven File-Processing Programming Model*, if you define an output bucket as the input bucket of another function, a workflow can be created. By doing this, AWS Batch and AWS Lambda executions can be combined through S3 events.

An example of this execution can be found in the video process example.

12.5 Limits

When defining an AWS Batch job have in mind that the AWS Batch service has some limits that are lower than the Lambda service.

For example, the Batch Job definition size is limited to 24KB and the invocation payload in Lambda is limited to 6MB in synchronous calls and 128KB in asynchronous calls.

To create the AWS Batch job, the Lambda function defines a Job with the payload content included, and sometimes (i.e. when the script passed as payload is greater than 24KB) the Batch Job definition can fail.

The payload limit can be avoided by redefining the script used and passing the large payload files using other service (e.g S3 or some bash command like 'wget' or 'curl' to download the information in execution time). As we did with the plant classification example, where a bootstrap script was used to download the executed script.

Also, AWS Batch does not allow to override the container entrypoint so containers with an entrypoint defined can not execute an user script.

12.6 Multinode parallel jobs

You can execute multinode parallel jobs in batch by enabling this mode either in the scar.cfg file or in the configuration file for the job (functions->aws->batch->multi_node_parallel->enable). You can also set the number of nodes and the index of the main node. Please take into account that the index of the main node starts from 0 up to the number of nodes -1.

We included an example of MPI job that can be executed as multinode parallel job, showing a hello world from each CPU/node available for execution. Both work in Amazon Lambda and Batch single node, you can use the included configuration files as a starting point. For more details, please check the README.md that comes with the example.

CHAPTER 13

Event-Driven File-Processing Programming Model

SCAR supports an event-driven programming model suitable for the execution of highly-parallel file-processing applications that require a customized runtime environment.

The following command:

```
cat >> darknet.yaml << EOF
functions:
  aws:
  - lambda:
      name: scar-darknet-s3
      memory: 2048
      init_script: yolo.sh
      container:
        image: grycap/darknet
      input:
      - storage_provider: s3
        path: scar-darknet/input
      output:
      - storage_provider: s3
        path: scar-darknet/output
EOF

scar init -f darknet.yaml
```

Creates a Lambda function to execute the shell-script yolo.sh inside a Docker container created out of the `grycap/darknet` Docker image stored in Docker Hub.

The following workflow summarises the programming model:

1) The Amazon S3 bucket `scar-darknet` is created with an `input` folder inside it if it doesn't exist.
2) The Lambda function is triggered upon uploading a file into the `input` folder created.
3) The Lambda function retrieves the file from the Amazon S3 bucket and makes it available for the shell-script running inside the container in the path `$TMP_INPUT_DIR`. The `$INPUT_FILE_PATH` environment variable will point to the location of the input file.

4) The shell-script processes the input file and produces the output (either one or multiple files) in the folder specified by the `$TMP_OUTPUT_DIR` global variable.

5) The output files are automatically uploaded by the Lambda function into the `output` folder created inside of the `scar-darknet` bucket.

Many instances of the Lambda function may run concurrently and independently, depending on the files to be processed in the S3 bucket. Initial executions of the Lambda may require retrieving the Docker image from Docker Hub but this will be cached for subsequent invocations, thus speeding up the execution process.

For further information, examples of such application are included in the examples/ffmpeg folder, in order to run the FFmpeg video codification tool, and in the examples/imagemagick, in order to run the ImageMagick image manipulation tool, both on AWS Lambda.

13.1 More Event-Driven File-Processing thingies

SCAR also supports another way of executing highly-parallel file-processing applications that require a customized runtime environment.

After creating a function with the configuration file defined in the previous section, you can activate the SCAR event launcher using the `run` command like this:

```
scar run -f darknet.yaml
```

This command lists the files in the `input` folder of the `scar-darknet` bucket and sends the required events (one per file) to the lambda function.

Note: The input path must be previously created and must contain some files in order to launch the functions. The bucket could be previously defined and you don't need to create it with SCAR.

The following workflow summarises the programming model, the differences with the main programming model are in bold:

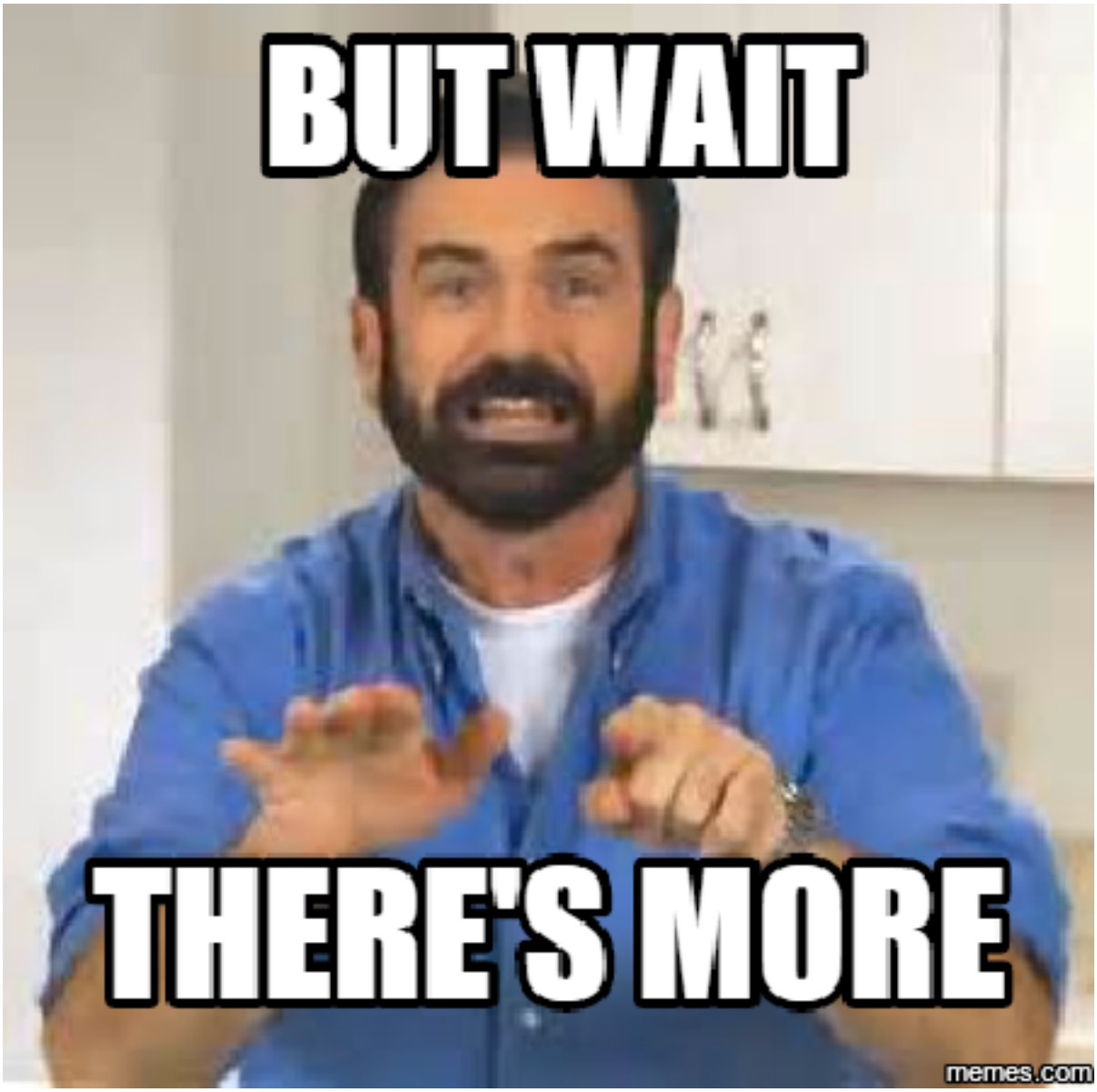
1) **The folder 'input' inside the amazon S3 bucket 'scar-darknet' will be searched for files.**

2) **The Lambda function is triggered once for each file found in the folder. The first execution is of type 'request-response' and the rest are 'asynchronous' (this is done to ensure the caching and accelerate the subsequent executions).**

3) The Lambda function retrieves the file from the Amazon S3 bucket and makes it available for the shell-script running inside the container. The `$INPUT_FILE_PATH` environment variable will point to the location of the input file.

4) The shell-script processes the input file and produces the output (either one or multiple files) in the path specified by the `$TMP_OUTPUT_DIR` global variable.

5) The output files are automatically uploaded by the Lambda function into the `output` folder of `scar-darknet` bucket.



13.2 Function Definition Language (FDL)

In the last update of SCAR, the language used to define functions was improved and now several functions with their complete configurations can be defined in one configuration file. Additionally, differente storage providers with different configurations can be used.

A complete working example of this functionality can be found here.

In this example two functions are created, one with Batch delegation to process videos and the other in Lambda to process the generated images. The functions are connected by their linked buckets as it can be seen in the configuration file:

```
cat >> scar-video-process.yaml << EOF
functions:
```

(continues on next page)

(continued from previous page)

```
  aws:
  - lambda:
      name: scar-batch-ffmpeg-split
      init_script: split-video.sh
      execution_mode: batch
      container:
        image: grycap/ffmpeg
      input:
      - storage_provider: s3
        path: scar-video/input
      output:
      - storage_provider: s3
        path: scar-video/split-images
  - lambda:
      name: scar-lambda-darknet
      init_script: yolo-sample-object-detection.sh
      memory: 3008
      container:
        image: grycap/darknet
      input:
      - storage_provider: s3
        path: scar-video/split-images
      output:
      - storage_provider: s3
        path: scar-video/output
EOF

scar init -f scar-video-process.yaml
```

Using the common folder `split-images` these functions can be connected to create a workflow. None of this buckets or folders must be previously created for this to work. SCAR manages the creation of the required buckets/folders.

To launch this workflow you only need to upload a video to the folder `input` of the `scar-video` bucket, with the command:

```
scar put -b scar-video/input -p seq1.avi
```

This will launch first, the splitting function that will create 68 images (one per each second of the video), and second, the 68 Lambda functions that process the created images and analyze them.

CHAPTER 14

Local Testing

14.1 Testing of the Docker images via udocker

You can test locally if the Docker image will be able to run in AWS Lambda by means of udocker (available in the *lambda* directory) and taking into account the following limitations:

- udocker cannot run on macOS. Use a Linux box instead.
- Images based in Alpine will not work.

Procedure for testing:

0. (Optional) Define an alias for easier usage:

```
alias udocker=`pwd`/lambda/udocker
```

1) Pull the image from Docker Hub into udocker:

```
udocker pull grycap/cowsay
```

2) Create the container:

```
udocker create --name=ucontainer grycap/cowsay
```

3) Change the execution mode to Fakechroot:

```
udocker setup --execmode=F1 ucontainer
```

4) Execute the container:

```
udocker run ucontainer
```

5) (Optional) Get a shell into the container:

```
udocker run ucontainer /bin/sh
```

Further information is available in the udocker documentation:

```
udocker help
```

CHAPTER 15

License

Apache License

Version 2.0, January 2004

http://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other

modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.
6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.
7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.
8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.
9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

To apply the Apache License to your work, attach the following boilerplate notice, with the fields enclosed by brackets "{}" replaced with your own identifying information. (Don't include the brackets!) The text should be enclosed in the appropriate comment syntax for the file format. We also recommend that a file or class name and description of purpose be included on the same "printed page" as the copyright notice for easier identification within third-party archives.

Copyright 2018 GRyCAP - I3M - UPV

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either

express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 16

Need Help?

If you have any trouble please email products@grycap.upv.es