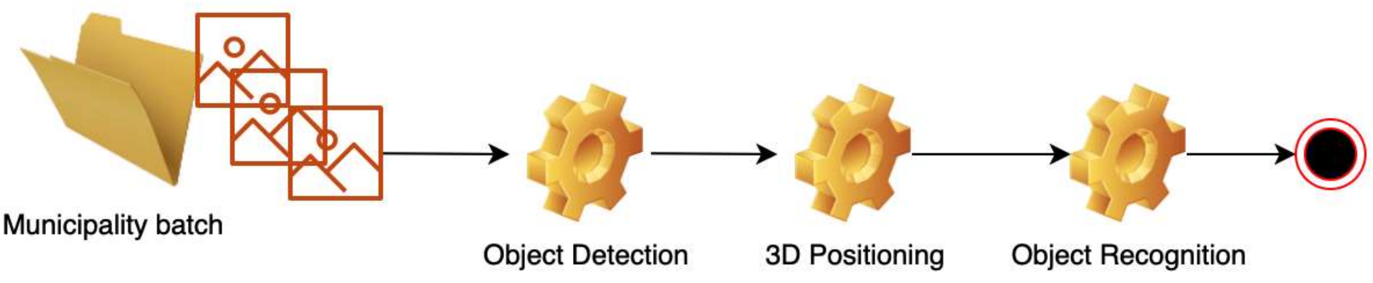
Readme.md

Assignment 1 - System Architecture

Premisses and assumptions

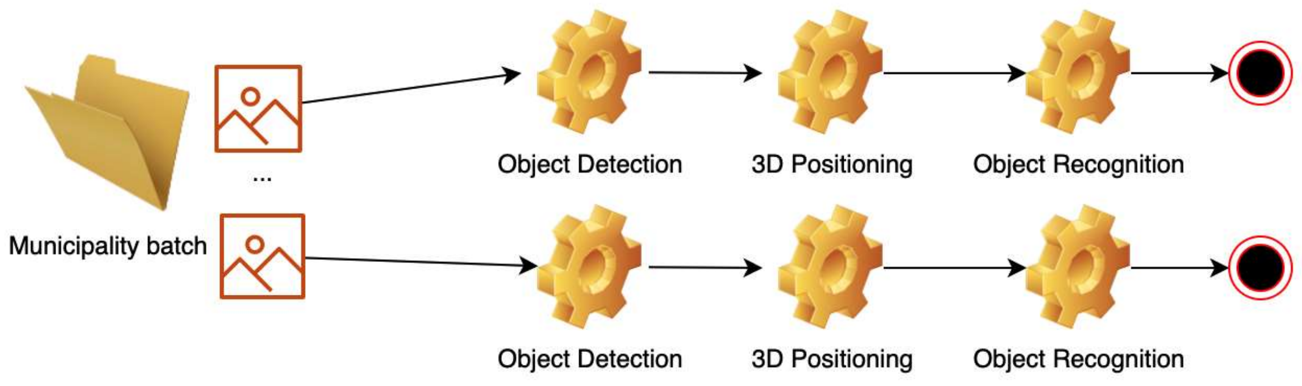
This architechture assumes that this data pipeline follow roughly this sequence diagram:



For each image in a municipality batch, execute a sequence of tasks based on the raw images.

Paralelization and algorithm characteristics

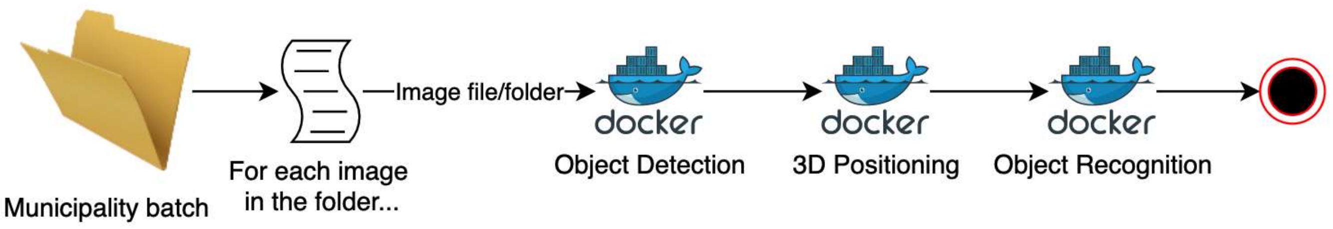
Since each Municipality has a list of images and, for each image, the whole pipeline is executed, each image may be processed in a single thread of the processing pipeline, hence:



Since the steps are naive about the state or origin of the previous steps, each step is scalable in terms of how many instances are available. Receiving N messages from the last step and processing acording to the instance number or resources available is possible.

Architecture 1 - Fully serial local processing

Run the entire pipeline for each image, in the same host, using a coordinator to list images and dispatch workload:



Coordinator runs a single "docker run" for each step.

The inter process communication may be simple stdout/stdin between dockers.

The format of the call may be simple json string.

Pros

- Simple, easy to implement and deploy

Cons

- Won't scale
- Host has to have all resources available and concentrated, including GPU
- Ultra high latency per Municipality
- Won't fit a real world large scale processing scenario
- **This is not a recomended way of doing this**

Improvement possibility

- Coordinator may run several parallel instances of the pipeline, depending on the server CPU/GPU count availability

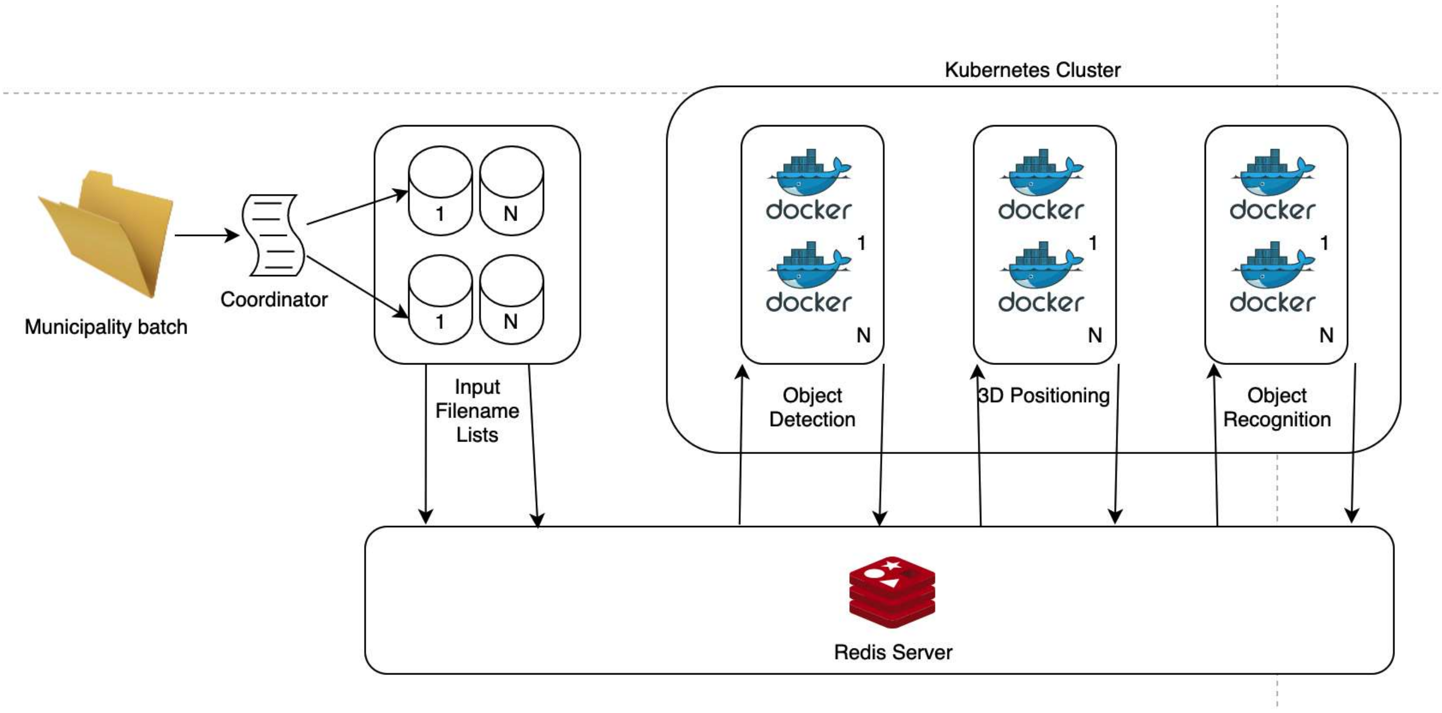
Architecture 2 - Multiple Problem, Multiple Data

A message passing architecture may be usefull in a real world scenario: data items or chunks of data may be distributed into the pipeline. This is far more scalable and, therefore, latency can be reduced increasing amount of instances of each step of the pipeline.

A process coordinator may distribute the shards for processing and the availability of instances of each step should dictate processing latency.

Each step may have instance number controlled, so if a single step is bottlenecking the pipeline, system administrator can increase instance count.

Scaling can be done in autoscaling tools like Google Kubernetes Engine, specifying various maximum and minimum thresholds and GPU quota.



Pros

- Scalable, may be automatic or manual
- Agnostic, may be implemented using docker-compose in Swarm mode, Kubernetes Cluster, OpenShift, Flynn, AWS and Google Cloud Solutions
- Scaling may consider GPU or CPU bounded parts of the processing pipeline, allowing to put energy and money where is needed

Cons

- Cost: despite manual or automatic scalability, there must be an amount of instances available so the autoscaling tools based on resource usage can work
- Management: this approach assumes the system administrator has full control of the cluter, therefore management and monitoring must be hand made

Improvement possibility

- Specify a autoscaling technology and thresholds

Cost and Latency

This architecture should scale horizontaly in the pod level. The higher the pod count, the lower latency.

Increasing pod count should also bring cost up, since in a cloud service the cost is based in the instance type and count.

Final considerations

- Don't know if there is a specification error between step 1 and 2. The I/O interface doesn't fit between steps. Maybe I have not understood how the algorithm works and the questions I made for this tasks did not help. Hope I could check this out with you later
- There are several products in the market for ML and serverless pipelined processing in the market, Amazon's SageMaker and Fargate are examples.
- In a similar maner, those docker images could be in ECR and used as Lambda Containers or deployed in Beanstalk to have benefits of a more manaaged and serverless approach for the architecture above
- I definately believe that the more serverless I could be in this, the better in terms of cost reduction and latency management
- I appreciate the this test as a usecase for me to learn cloud tech and products I'm not familiar with so thank you anyway, this was fun!

Bibliography and study sources

- https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler
- https://cloud.google.com/kubernetes-engine/docs/how-to/gpus

- https://www.replex.io/blog/kubernetes-in-production-best-practices-for-cluster-autoscaler-hpa-and-vpa
- https://docs.aws.amazon.com/lambda/latest/dg/images-create.html
- https://aws.amazon.com/pt/blogs/machine-learning/building-a-scalable-machine-learning-pipeline-for-ultra-high-resolution-medical-images-using-amazon-sagemaker/
- https://github.com/aws-samples/sagemaker-distributed-training-digital-pathology-images
- https://aws.amazon.com/pt/blogs/architecture/field-notes-building-an-automated-image-processing-and-model-training-pipeline-for-autonomous-driving/
- https://aws.amazon.com/pt/solutions/implementations/aws-mlops-framework/
- https://github.com/awslabs/aws-mlops-framework
- https://github.com/awslabs/amazon-sagemaker-mlops-workshop
- https://github.com/aws-samples/mlops-amazon-sagemaker-devops-with-ml
- https://github.com/aws-samples/amazon-sagemaker-secure-mlops
- https://aws.amazon.com/pt/elasticbeanstalk/

Assignment 2 - Code

Premisses and assumptions

The code is based on the provided documentation:

[Assignment](docs/Assignment Cloud Software Engineer and Devops 20210809.pdf)

The goal of this API is to provide a flexible image processing tools, based on the requirements described in this document.

I took for granted some assumptions based in the instructions:

`Any combination of the above operations`

My assumption at this point is that the user may specify not only the combination, but the order of the parameters and processing. *The image processing is considered to be a pipeline where every step uses as input the output of the last step*, in the order the user provided.

`Resizing the image to any reasonable size smaller than the origin image.`

Also considered that the user may be free to resize the image as he likes. I could assume the resizing to be done in many different ways (percentual, fixed propotion by axis, totally flexible) but it was easy to make the API plain flexible and allow resizing anyway the user want. Also, as this processing is a pipeline, if the resizing comes after a file spliting, the splitted images shall be resized to the target size provided by the user.

`All images can be assumed to be present as local files on the same machine where the service runs or any location which makes your life easier.`

The service implementation allows the user to call the HTTP interfacing and, in the same request, send the image for processing as multipart. As this makes my life easier (and maybe the user's too), I have designed this way. Changing this to point to a location in the disk is also fairly simple in this API design. An usage example is also included in this repository.

For convenience the processed files are available in the server instance in a static file folder instructions below.

`You can assume that there will be only 1 request at the same time although more request may come in while the processing is still running.`

`You do not have to take into account the issue of dependencies between requests.`

The service may take concurrent calls, as the Celery worker may receive paralel requests and will process them as it appears. But, in fact, the application simplify the file tracking and concurrent calls with same file names may incur in some sort of race condition. As this tip allows me not to worry with that, I'll use in my favor.

`You can assume the height and width of all input images are 512x512 + 1024x1024 or any size limitation which is convenient for you, expect that the code should be able to support at least two different sizes.`

The API accepts arbitrary image, no need to constrain this. I guess this fills this requirement and makes a flexible API more... flexible :)

```
You could use flask as a web framework for you
```

```
You could use opencv to process images
```

Chose aiohttp over Flask because is... plain fun. Not very different, same approach for interfaces and application entrypoint/blueprints.

```
You're free to use any open-source distributed processing framework include one as simple "make"
```

I chose Celery+Redis to fulfill this requirement.

Installation

Linux/OSX

Dependencies:

- make
- Docker: Get Docker
- Docker Compose

```
$ pip install docker-compose
```

Clonning and Running:

```
$ git clone https://github.com/SamambaMan/imageprocessing.git
$ cd imageprocessing
$ make run
```

Daemonize:

After clonning

```
$ make daemon
```

And to stop daemon:

```
$ docker-compose stop
```

Testing:

```
$ make test
```

Development Console:

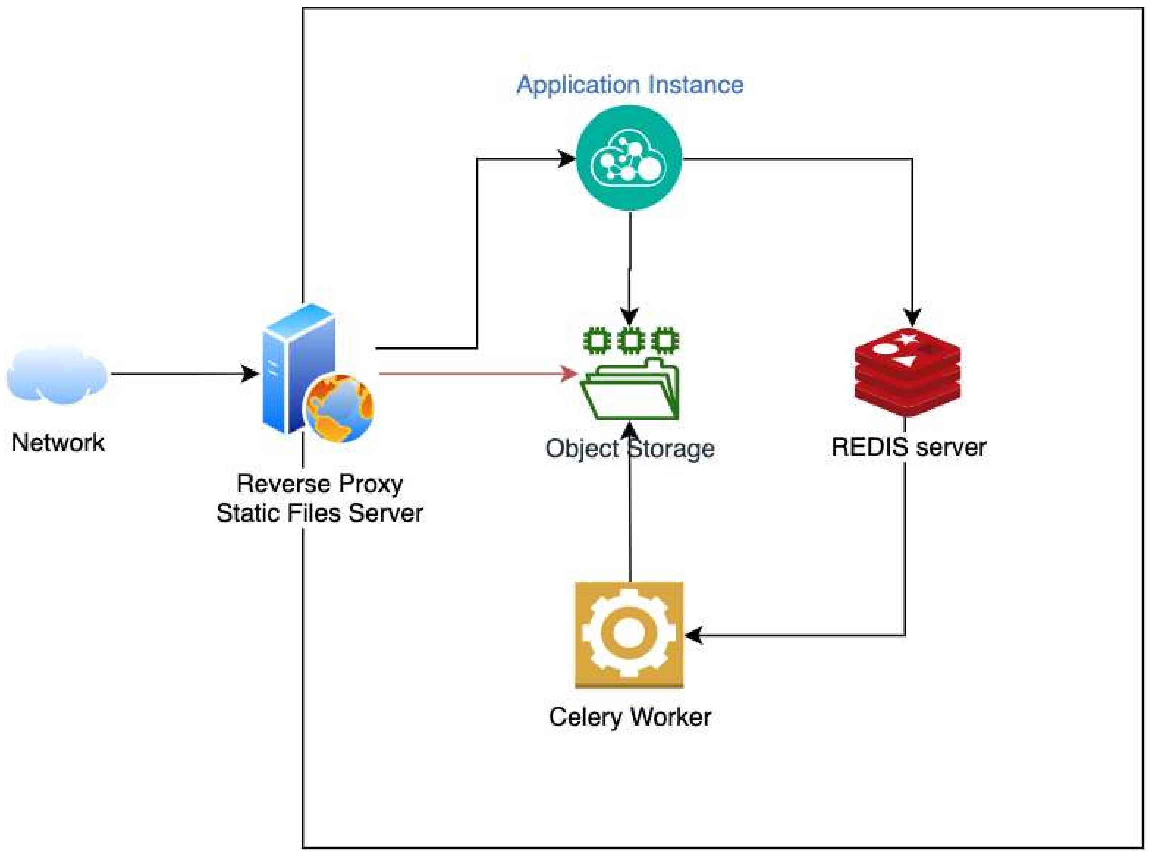
```
$ make devconsole
```

Debugging While the application is running, developer may want to debug using python pdb or similar. If so, run another terminal instance and:

Web Container:

```
$ docker attach $(docker ps | grep imageprocessing_web | cut -f 1 -d ' ')
```

Application architecture



The application uses a simple architecture, segregating internet networking from the application infrastructure through NGINX.

The web server provides a reverse proxy for the application instance and also works as a simple HTTP file server so the user may download the files from the storage instance.

Service Interface

```
localhost:8080/process - method: POST
```

The api accepts a multipart/form-data post with two fields:

- actions - JSON process parameters
- files - *array* of JPG images

Static File Server

After image processing, the user may check sent and processed images in:

```
http://localhost:8080/static/
```

Actions JSON template

```
{
    "operations": [
        ["resize", "200x300"],
        ["split", true],
        ["blur", "gaussian|updownsampling|bilateral|median"]
    ],
    "output":"jpg|png"
}
```

The user may specify the parameters in any order, as the API will process them in that specified order.

Call example

First of all, install aiohttp for requests:

```
$ pip install aiohttp
```

Then, run the following code as a call example:

In the source foder, user may execute the following Python code (I recomend ipython):

```python
import aiohttp
from multidict import MultiDict

postdata = """{
        "operations": [
            ["resize", "200x300"],
            ["split", true],
            ["blur", "gaussian"],
            ["split", true],
            ["blur", "gaussian"],
            ["resize", "600x600"],
    ["blur", "gaussian"],
    ["split", true]
        ],
        "output":"jpg"
    }"""

async with aiohttp.ClientSession() as session:
    datacontent = MultiDict([
        ('actions', postdata),
        ('files', open('app/tests/fixtures/test-pattern.jpeg', 'rb')),
        ('files', open('app/tests/fixtures/tv-pattern.png', 'rb'))
    ])

    resp = await session.post(
        'http://localhost:8080/process',
        data=datacontent
    )

    response = await resp.content.read()

    assert response == b'OK'
```

Improvements:

- Better integration tests: this POC doesn't include any integration tests and would be nice to have that accounted. Would be a nice oportunity to use pytest-docker-compose.
- Better HTTP method signatures: I could have included serializers for the HTTP interface in order to expose to the user automated API tools.
- Tests: I miss some error testing, forcing the code to raise exceptions and check if it's done
- Request validation: this application don't include any request validation to make the user's life easyer. A 500 error and that's it