CURART

Technical Manual

CURART: Stolen Image Detection

Thomas Doyle - 15350316 - 19-05-2019

Technical Manual

Contents

Introduction

Overview and Motivation

This project is written in python 3 to attempt to identify images of artworks. The purpose of this is to attempt to find people who are using copyrighted material inside of images even if they are modified from the original. This is implemented using the OpenCV version of the SIFT algorithm. Therefore it is not for commercial use and purely a research project. Because this is a research project, the implementation is less about the identification of the artwork but more an exploration into the identification of complex objects within images, that may be obscured.

While the domain of this project is restricted here to the domain of just artworks, this project can easily be used for other purposes of finding images that are made up of the exact same content. It will not use context at all for matching, i.e a different angle of the same building. For this to match it is a goal of this project to only match if the images are the same.

The second aim of the project is to work at scale. If this tool is to be usable the user should not have to wait hours to find a result. This research project should be able to get matches in a reasonable amount of time.

Research

The first point of research for this project was to find what research has already been done in the area. This was gone about by researching academic papers in the area on IEEE. This provided a lot of use, even though papers were generally dense with content and quiet hard to understand and make sense on sometimes because of the lack of knowledge I had going into the project, and in relation to the field of computer vision and graphics processing still don't. But we what I do know have is an understanding of feature identification and search.

Through my investigation of previous work I found a number of articles using the SIFT (Scale Invariant Feature Transformation) algorithm to identify key points in their images and then using them key points to identify features inside the image [1,2,3]. This gave me the idea to use the SIFT algorithm to get key points and "fingerprint" and image by getting the key points. I could then save these key points in a database and match against them at a later date.

I also explored other less peer reviewed option to see if there were any new and emerging technologies that had not made it into publications. This is where I found out about the histogram approach of matching images [4]. But this was a worse

solution because it could be fooled by cropping, flipping and warping. It would also fail on watermarks, hues and if the image was inside another.

While researching more about SIFT I came across a paper about Semantic Texton Forests for Image Categorization and Segmentation [5,6]. This was using key points in an images to get textures and use many many decision trees called decision forests to classify images. At the time this was a far more complex method to wrap my head around. I also came to the conclusion that classifying the image was not the goal and we wanted to find matches. This made the speed increase from using this method irrelevant.

In this project we set out to build on this research by seeing if these key points could be used at scale to match identical images.

Context Diagram

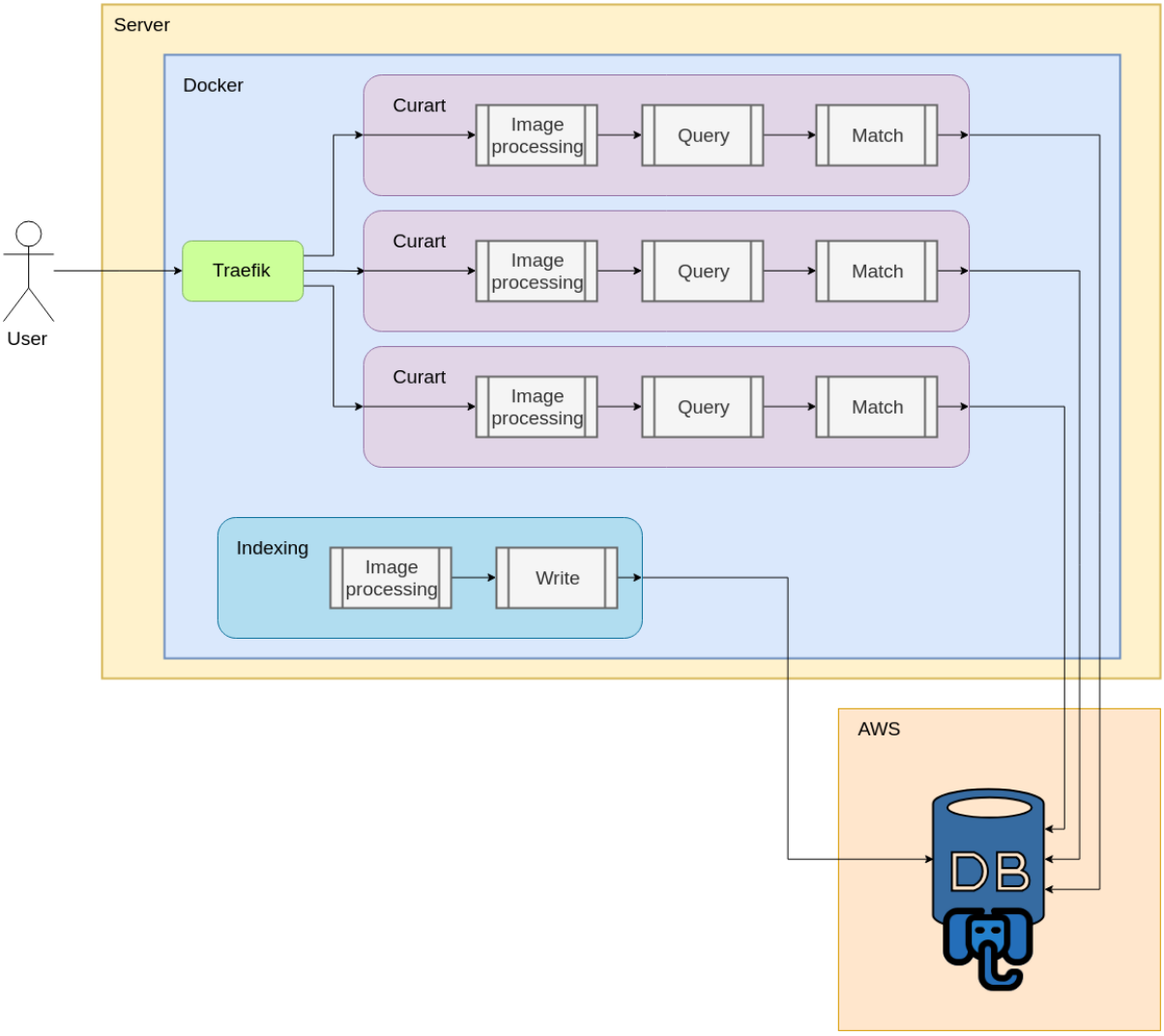


Figure 1.1 *System context diagram*

Glossary

AWS: Amazon Web Services is a subsidiary of Amazon that provides on-demand cloud computing platforms on a metered pay-as-you-go basis.

Container: A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

Docker: A tool designed to make it easier to create, deploy, and run applications by using containers. (Any reference to containers, you can presume that these are docker containers unless otherwise stated.)

EC2: Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services in the form of virtual servers.

OpenCV: A library of Python bindings designed to solve computer vision problems

Traefik: A modern HTTP reverse proxy and load balancer that makes deploying microservices easy.

PostgreSQL: A powerful, open source object-relational database system

S3: Amazon S3 has a simple web services interface that you can use to store and retrieve any amount of data, at any time, from anywhere on the web.

System Architecture

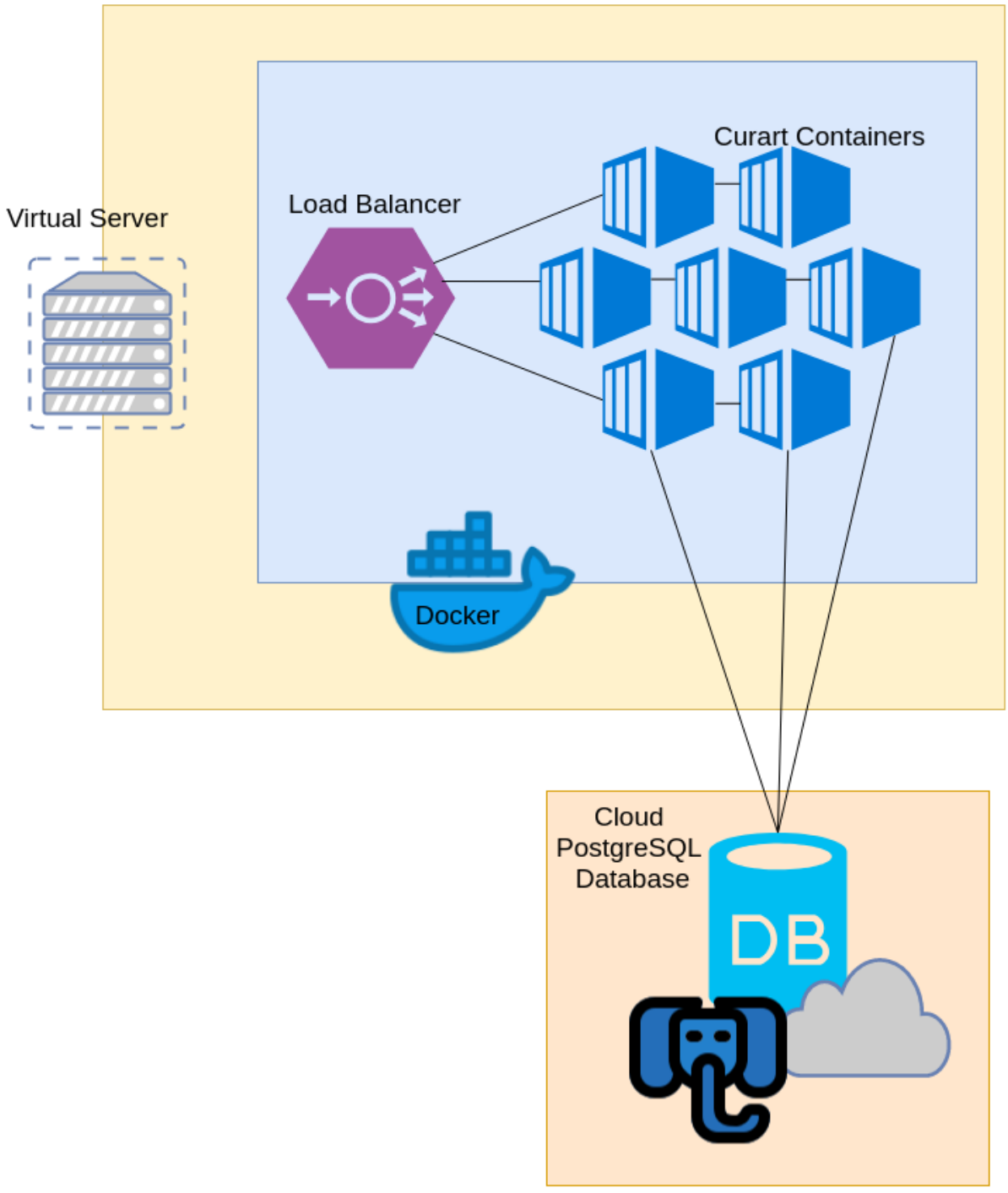


Figure 2.1 *System architecture diagram*

UI Design

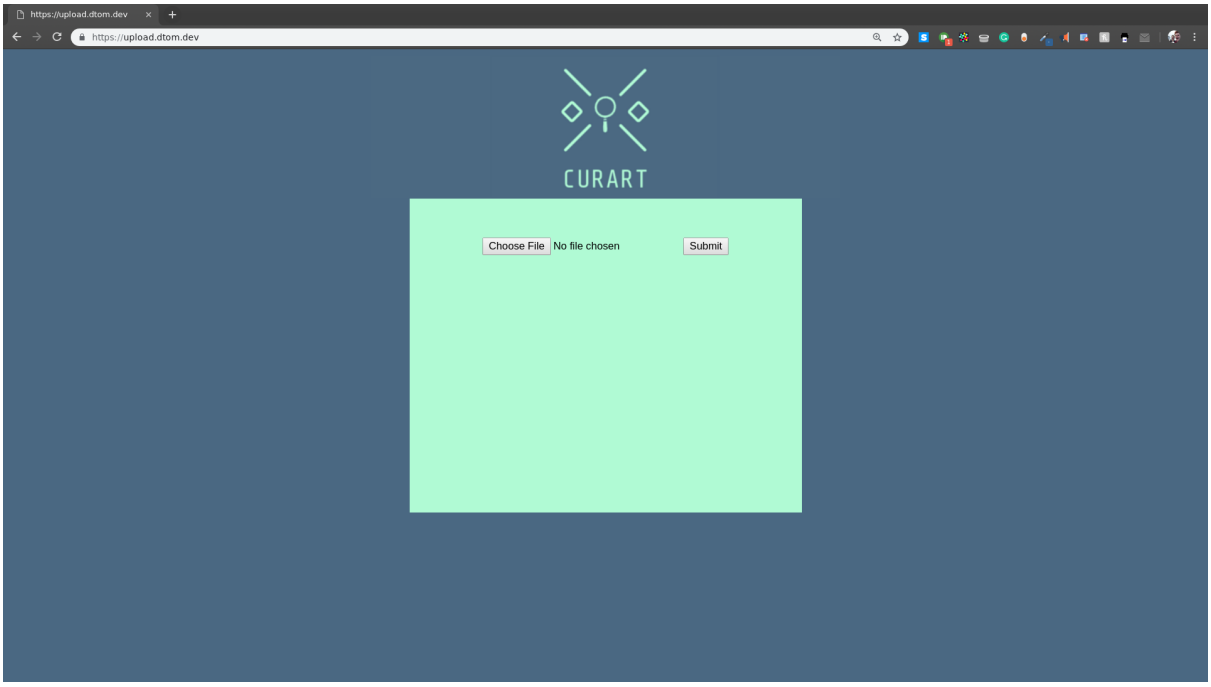
The user interface has been designed to be very minimal with only two buttons. This was done to emulate the big search engine applications that people are already familiar with. There is no help functionality on the interface because, with the minimal buttons and design, it may only add to confusion with an already intuitive design.

The color scheme is darker in the edges and brighter in the center to direct focus toward the functionality of the system. The colors also remain in the color scheme of the logo and project as a whole.

When the file is processing a loading screen clearly is moving to indicate to the user that there is processing happening in the background.

The buttons are clearly labeled and are the center focus of the application. The button also provides a filename preview of what you are about to upload to the application.

After image processing the image is returned in the center of the screen and a back button clearly label underneath.



Docker Image

The container needs an image to run on. Originally I was using the Ubuntu image and installing everything every time it was run. This was very slow when I started using OpenCV and it had to be compiled every time. This was also exacerbated in the Gitlab CI pipeline. It was then realised early on that a custom image could be created that is based on the Ubuntu image but with a dependencies built in. This sped up testing and deployment time for over an hour to seconds.

Below is the Dockerfile. It has minimal `Run` commands because each RUN is a new container layer, lengthening build times. It also removes files as it goes to reduce the size of the image once built to have a small design.

```
FROM ubuntu:latest

MAINTAINER Tom Doyle <thomas.doyle9@mail.dcu.ie>

ENV DEBIAN_FRONTEND=noninteractive
RUN apt-get update && apt-get install -y \
    build-essential \
    cmake \
    rsync \
    git \
    pkg-config \
    wget \
    python3-dev \
    libopencv-dev \
    ffmpeg  \
    libjpeg-dev \
    libpng-dev \
    libtiff-dev \
    opencv-data \
    libopencv-dev \
    libgtk2.0-dev \
    python3-numpy \
    python3-pycurl \
    libatlas-base-dev \
    gfortran \
    webp \
    libavcodec-dev \
    libavformat-dev \
    libswscale-dev \
    libv4l-dev \
    libatlas-base-dev && apt-get clean

RUN wget https://bootstrap.pypa.io/get-pip.py && python3 get-pip.py && \
    pip3 install numpy && \
    rm -rf get-pip.py
```

```
RUN git clone git://github.com/trendmicro/tlsh.git && \
    cd tlsh && git checkout master && ./make.sh && \
    cd py_ext/ && python3 ./setup.py build && python3 ./setup.py install &&
\
    cd ../.. && rm -rf tlsh

RUN cd ~/ && git clone https://github.com/opencv/opencv.git && \
             git clone https://github.com/opencv/opencv_contrib.git && \
             mkdir -p ~/opencv/build && cd ~/opencv/build && \
             cmake -D CMAKE_BUILD_TYPE=RELEASE \
                   -DCMAKE_INSTALL_PREFIX=/usr/local \
                   -DOPENCV_EXTRA_MODULES_PATH=~/opencv_contrib/modules \
                   -DBUILD_opencv_python3=ON \
                   -DPYTHON_DEFAULT_EXECUTABLE=/usr/bin/python3.6 \
                   -DOPENCV_ENABLE_NONFREE=ON \
                   -DBUILD_EXAMPLES=ON .. && \
             make -j`cat /proc/cpuinfo | grep MHz | wc -l` && \
             make install && ldconfig && \
             rm -rf ~/opencv && rm -rf ~/opencv_contrib
```

Language Choice

Python is an easy to develop language but has some speed impacts when it comes to heavy computation such as images. But after research into the SIFT algorithm I saw others that had successfully implemented SIFT with python with minimal impacts to speed. Source It became apparent that implementing SIFT was not new and had been done many times before. So rather than reinventing the wheel I wanted to add to work that has been done before me.

This leads to the research into previous implementations of SIFT algorithm. The OpenCV library's implementation was one that came up again and again as the most optimised and accurate. This is the one chosen and is written in C with a python interface so this allows us to use it with python.

Python has many packages for web frameworks that many other low level languages do not. This also made it ideal that we could use it for many different uses throughout the architecture of the project.

Python is also the language I am most comfortable in, and one of the non functional requirements of this project is to meet the project deadline. There for this is the language I can get all of the features I want to add into this project in time for the final deadline.

Python2.7 was not considered because the end of life for this version is in 8 months at time of writing.

All these points resulted in me choosing python 3 for this project.

High-Level Design

Initial Design vs Current Design

The initial design of this project has changed significantly since it was first conceived. This was due to lack of experience in using the technologies and also unforeseeable issues that were encountered. We will talk about the differences in this section and we will talk about the problems encountered and why the design has changed in a later section.

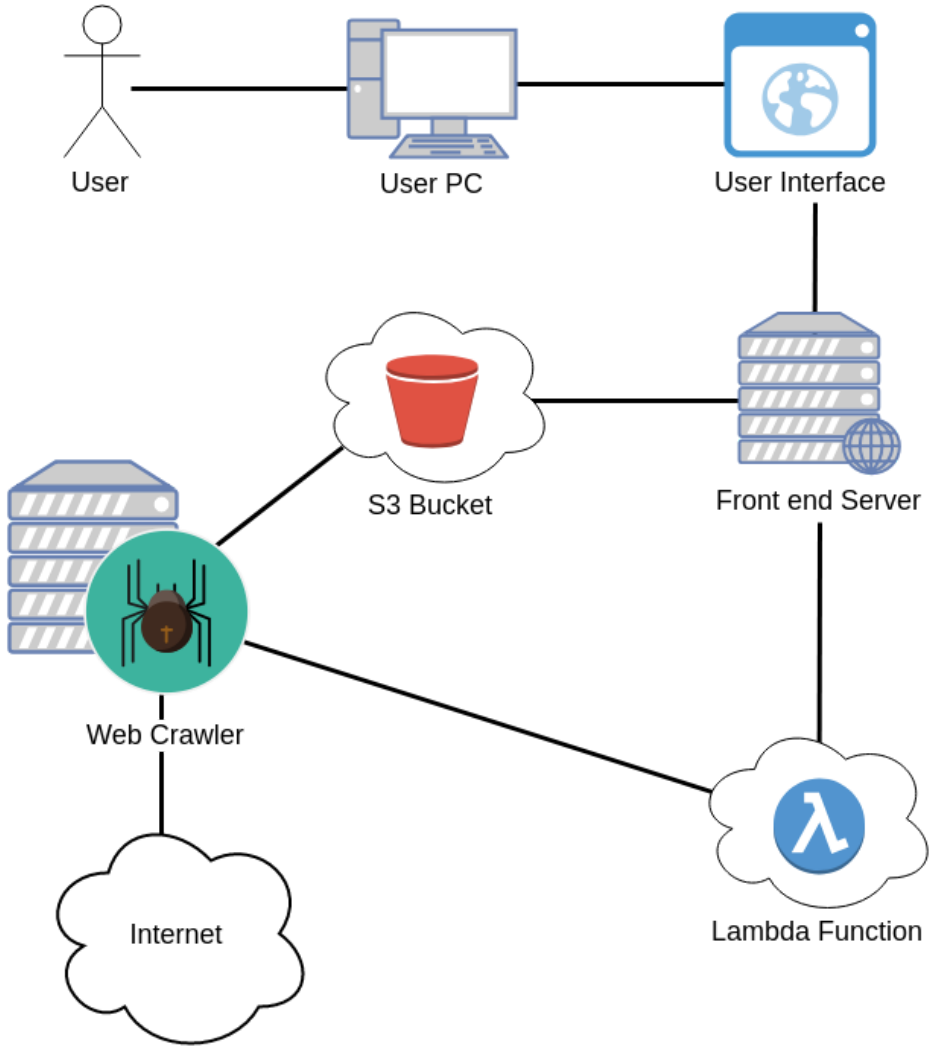


Figure 3.1 *Original design of the project*

The first major design change in the lack of AWS lambda functions. These have been replaced by docker containers that are load balanced by a software defined load balancer called Traefik. This will allow the same amount of scalability but does mean that the base server will have to have to be of sufficient size to handle the traffic rather than a small server that can hand off the computation to another compute module.

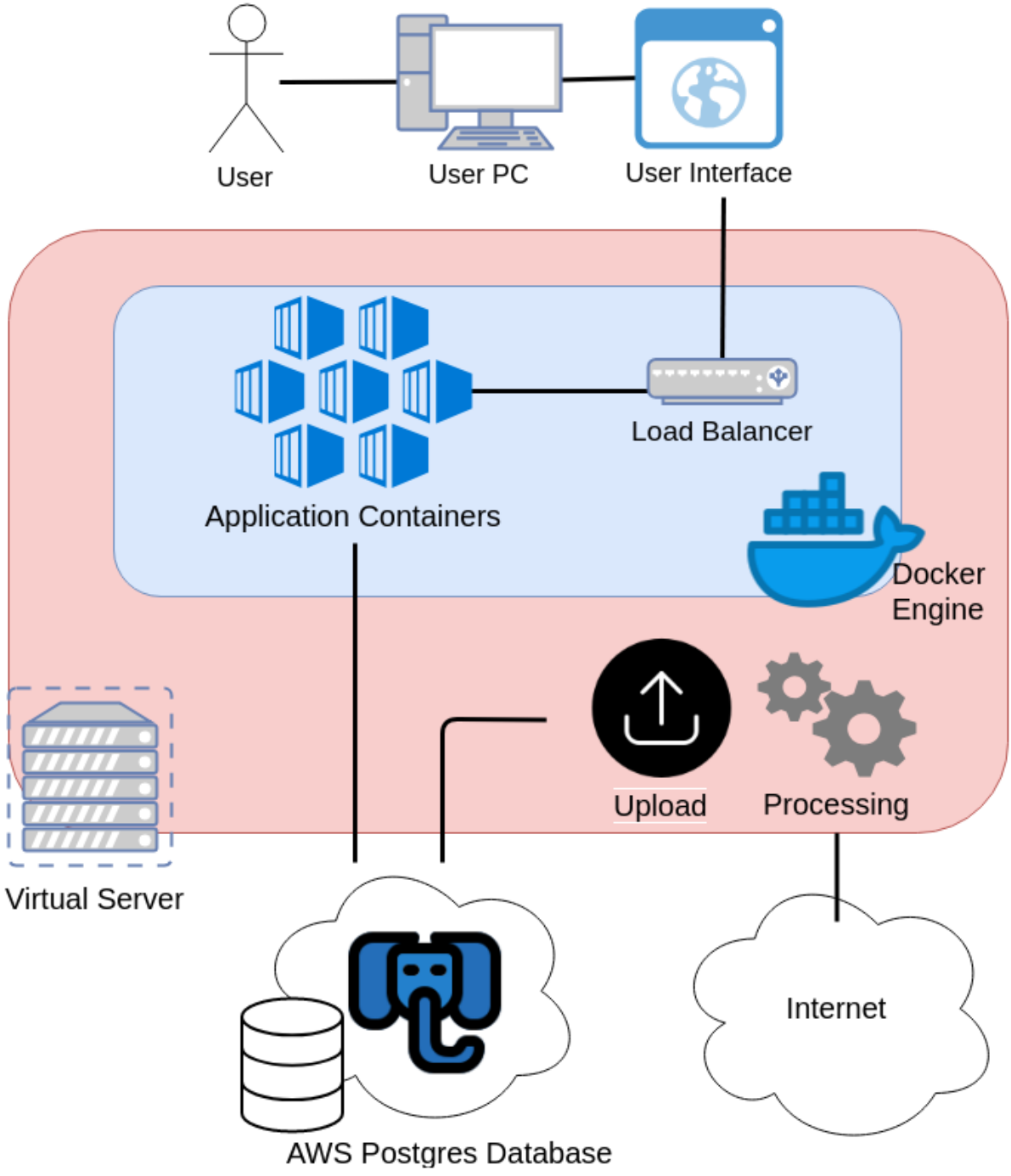


Figure 3.2 *Updated design of the project*

The datastore for this project as also changed from an amazon S3 bucket to a PostgreSQL database hosted by AWS. This pivot was due to the purpose of S3 buckets being an object store and unstructured data. This was a slow design that would involve searching the whole database if you were looking for a best match like we are rather than retrieving known data. A better solution was to hash the keypoint object on the way into a PostgreSQL database and create a structured form for the data. Then on query we use the Levenshtein distance on the hash to find the best match quickly. This still provides us with the reliability and availability of amazon services which was desirable at the beginning of this project. Postgres also has a Levenshtein function built in. So rather than retrieving all hashes and finding the best match an querying again, we can do the calculation closer to the data speeding up the data retrieval significantly.

The web server has also now been containerised to run multiple instances on the server rather than one for all incoming traffic. Each container handles its traffic internal. The Traefik load balancer for the containers passes the traffic to the container in a round robin method.

Problems and Resolution

OpenCV Issues

Throughout the project there were a number of problems that turned into big learning points. The first was the attempt to learn a new technology and use the OpenCV library. It became clear from my research that SIFT was the algorithm choice for this design and that OpenCV had the best implementation out there. The learning point came trying to learn this library. The SIFT algorithm is a non free algorithm in the sense it is patented to be only used in research projects. To avoid a situation where a person would use this algorithm unknowingly, it was taken out of the packaged version of OpenCV in version 3.0.0. It meant that it had to be compiled in with the opencv-contrib modules. This was a learning curve to overcome because of system dependencies on python. After many attempts to install I got it working for python 2 and not python 3. The only solution to this was do a clean system install and compile on a new installation. This solved the problem and a blog was created for how to install OpenCV for python 3 on Ubuntu.

Lambda Issues

Another design problem that was faced was learning the technology lambda. Once the tutorials were done and I had a good understanding of lambda functions and how to use them I began trying to get dependencies, mainly OpenCV to run on a lambda.

Because this package has to be custom compiled I needed to compile this on a lambda machine. Amazon has Elastic Computing (EC2) instances that replicate the lambda environment for this purpose but the compilation runs out of memory in this restricted environment. The compilation time was reduced but removing c files from the library by hand but still, the lambda was missing shared files that OpenCV depended on. From blogs and online forums I saw it was possible to run OpenCV on a lambda but I did not see anyone able to run OpenCV with contrib modules compiled in. This forced a shift in design. A custom docker image was created and the application was containerised with docker. This allows multiple applications to run in parallel and be load balanced, giving it similar scalability in the original design.

Database Redesign

The original design also made use of an S3 bucket to store key point objects in. After learning more about AWS is became apparent that S3 was not a perfect solution and in fact it would be rather slow to use in this application. It's primary use is unstructured data, making it great for backing up small and large files and objects. But the unstructured nature makes it hard to find things quickly when you don't know what the exact thing you are looking for is, e.g. in this case a best match search. The solution here was to change out the database for a PostgreSQL database. This allowed us to create a locally sensitive hash for each key point and do a Levenshtein distance measurement on it to get a best match quickly.

Re Scoping of Project

In the original design it was proposed that the system would find and search the internet for images and preprocess them to find image across the internet. This became a problem when the domain of the project was changed from any image across the internet to artworks. This added more complexity to only allow in images that were of artworks and not add images of people, places or other domains. It was decided that this was out of scope to be able to do. Images can be added semi autonomously by feeding a file of urls to the image adding module. This will handle the downloading, preprocessing and storing of the image key points.

Deployment

Docker-compose config file. Below is the docker-compose file to deploy the system. This will start two containers. One Traefik container that will route all the traffic to the containers and load balance between then, and a CURART application container.

To deploy copy file and run `docker-compose up -d`

To scale up or down the applications run `docker-compose up -d --scale curart=$instances_required`

Note: You will have to change the `traefik.frontend.rule` for both images to your own endpoints.

```
version: '3'

networks:
  proxy:
    external: true
  internal:
    external: false

services:
    curart:
        restart: always
        build:
            context: .
        environment:
            DBUSER: ${DBUSER:-latest}
            DBPASSWORD: ${DBPASSWORD:-latest}
            DBHOST: ${DBHOST:-latest}
            DBNAME: ${DBNAME:-latest}
            DBPORT: 5432
        labels:
            - traefik.backend=upload
            - traefik.frontend.rule=Host:upload.dtom.dev
            - traefik.docker.network=proxy
            - traefik.port=8080
        networks:
            - internal
            - proxy
    traefik:
        image: traefik:alpine
        container_name: traefik
        restart: always
        command: --docker
        ports:
            - "443:443"
            - "80:80"
        networks:
            - proxy
        volumes:
            - "/var/run/docker.sock:/var/run/docker.sock:ro"
            - "/etc/containers/traefik/traefik.toml:/traefik.toml:ro"
            - "/etc/containers/traefik/acme.json:/acme.json:rw"
        labels:
            - "traefik.frontend.rule=Host:monitor.dtom.dev"
            - "traefik.port=8080"
```

Testing

Unit and regression tests were run with a Gitlab pipeline. This pipeline had three stages.

Stage One Testing

This stage installed small requirements to the image for testing and then runs tests, located in a test folder. Any files beginning with test_ will be run. A coverage score is then reported in the pipeline. Only when this stage passes can the pipeline move onto the next stage.

Stage Two Deployment

This stage will then transfer files to the production server and redeploy the containers. This stage will also notify the slack development channel that it has done so.

Stage Three Deployment Testing

This stage tests the deployment has not been taken offline or has caused errors. If this is the case this stage will find the last working version from the Gitlab API and attempt to revert to that version and deploy that version. It will then send an alert to the slack development channel that it needs assistance with the error further.

Unit Testing

- Once a project is committed to the master branch a Gitlab pipeline is run.
- The first step in this is unit testing.
- PyUnit was used to run these tests and coverage was used to measure the coverage of these tests.
- The current status of these test are:

- This pipeline does only run on the master branch but could be expanded to run on any branch
- Committing to master branch is not ideal because it sometimes becomes unstable for a short amount of time
- Despite this it is felt because there is only one developer on this project it makes sense in this context

- The image that was used for testing is a custom image written by this development team for this project. It's files can be found here
- It has its own deployment and will automatically be uploaded to Dockerhub once the image it built
- This was necessary because the image takes over an hour to build. This includes installing all the packages necessary to run this application
- In the pipeline OpenCV is compiled and installed with extra modules
- The deployment pipeline for this project could not be run on Gitlab because it takes an hour to build and Gitlab runner timeouts are set to 10mins
- To get around this Github was used to host the source and Circleci was used to build the project

Post Deployment Testing

- The final stage in the pipeline is to test if the deployment succeeded and it running in production environment
- If this test fails it triggers and alert that is sent to a slack channel
- There are no other alerts channels in place because they were deemed out of scope for the context of the project and timeline but is on the backlog for future work
- In the event the deployment fails the system will attempt to pull from the Gitlab API to find the commit of the last successful build
- It will the attempt a git reset and redeploy the last working version and alert the team through the slack channel it has attempted this

User Testing

- For this project no real users we asked to complete a test of the application. Due to the limited user interaction function and intuitive design.
- This does not mean that there was no user testing done on the application. To verify this application, each user story was taken and a verified the functionality laid out in this document was easy and intuitive. No issues were found while doing these steps and it is believed that this would satisfy a users needs.

Functional Testing

Each functional requirement was taken and after a stepping through the application it was verified if the task could be completed

Must correctly identify image as artwork
Must identify artwork as similar despite being slightly altered
Must be able to return a result within a defined reasonable time period (reasonable was defined as 10 seconds)
Must be able to find images unsupervised
Must be able to store metadata about images not the image

During the testing it was found that one requirement was not met. This was removed because of the complexity around unsupervised web scraping within the domain. This requirement was thought of out of scope within the domain

Non Functional Testing

Time

The project was to be completed by the 19th of May 2019 at 23:59 and that has been completed

Ease of Use

This tool has been created with a minimal and intuitive design. It has taken methods and minimal styles like other modern search engines to have a familiar look and feel.

Storage

The storage on this project is hosted by AWS meaning that it is fast and reliable. It has also been small enough to stay on free tier, so it has not hurt the cost constraint.

Costs

The cost of the virtual server is 4 euro a month. Well within a reasonable price.

Hardware

The hardware has been kept minimal with 2GB of RAM, 1 2 GHz VCore and 20GB SSD. This means that the cost can be kept low

Speed

Image Size	Test One	Test Two	Test Three
67.4kB	5.34s	5.20s	5.42s
152.0kB	8.33s	8.18s	8.56s
109.9kB	9.90s	9.71s	9.52s
443.7kB	10.36s	10.11s	10.16s

- These speed I am confident can be improved on with the use of cache between the application and the database
- Currently the database is hosted in Ohio on AWS on a preview server
- A lot of the key points are large and similar quit often
- While there is cache on the page AWS only allows access to the database through their domain and Cloudflare could not be used as a simple fix
- This is achievable to get AWS cache service working but out of scope for this iteration of the project.
- It will remain on the backlog for future work

Algorithms

SIFT (Scale Invariant Feature Transformation)

Why use SIFT?

The requirements laid out at the start of this project in the Functional Spec said that for the requirement to be satisfied an accurate result would have to be returned in under 10 seconds. This was a big feat to attempt to accomplish and after much research SIFT was an algorithm that came up again and again in papers.

It was satisfactory algorithm in that it was ignorance of differences in:

- color
- size
- rotation

- perspective
- watermarks
- hues

For this reason it was thought of as the perfect solution to this problem if utilized correctly.

The next major decision around this algorithm was whether this project would implement it's own or if it would use a library for it.

After reading the work involved in implementing this in python, a quick development language we were not sure if we could fulfill the rest of the requirements in the project. There were also a good few examples of other people doing have done this open source already. Therefore the decision was made to use a library for the functionality

One library came up again and again and that was OpenCV. This has a C implementation with a python interface which give all the speed of python with the ease of integration of python. This was also recommended as a previously heavily optimised version, developed over a number of years, rather than function in a college project.

This function is an extra module in OpenCV and needs to be compiled in. In this project, that is done in the custom docker image we created with OpenCV and python inside. Feel free to use that image or if you want to compile your own OpenCV check out my blog on how to do that.

How does SIFT work?

1. Scale Space Construction
    - This is where the image is progressively blurred out more and more to have images that range from full detail to little detail in them. Then a range for each blurred image is created of different scaled images to create a scale space of several images.
2. LoG Approximations
    - We now want to make this scale space useful. We do this by attempting to find key points in it. A good way to do this is blur the image and then calculate second order derivations on it or the "laplacian". This operation is called the Laplacian of Gaussian. It is extremely sensitive to noise but the blur on the images helps this. Another problem is calculating second order derivations in computationally expensive. To generate Laplacian of Gaussian images

quickly we use the scale space by calculating the difference between two consecutive scales, or the difference of two Gaussians.

3. Finding Keypoints
    - A keypoint is marked as a maxima if it is the greatest of all it's neighbors including the image above and below it in the scale. The same is done for the minima. These are approximate minima and maxima because the real minimal and maxima almost always mathematically falls between pixels. Subpixel minima/maxima is done by getting the Taylor expansion of the image.
4. Remove Bad Key Points
    - A lot of keypoints can be generated in the previous step and many of these are of low contrast or lie on an edge. In this case they are not much use so they can be discarded.
5. Assigning Orientations to Keypoints
    - We now have keypoints and their scale (we know this from the scale of the image they were found in). We want to find the orientation for the keypoints. For this we look to the neighbour around the key points for gradients and magnitudes.
6. Generating the Features
    - We want to be able to identify features so we can create a fingerprint of key points for a feature. We create a window around the feature. We then split up that windows into even segments and generate a gradient angle for each window then normalise all the values.

This is a complex algorithm but diagrams make it far more manageable. The source for the quick explination is adapted from
http://aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/

TLSH - Trend Micro Locality Sensitive Hash

This locality sensitive hash function allows the key points to be hashed before being put into the database. This allows the result of our query to be a subset of the database rather than the entire database. We can search based on the hash to find close matches to the hash and return the values that are close to it.

It is also open source, of which can be found here on tlsh github

Database Search

For the database search we wanted to search based on how close the hash was. For this we used the Levenshtein distance between the hash in the query and the hash in the database. This mean that is can be a fast match rather than pulling back the

whole database. This also means that we can tune the query to come back with more or less accurate results. The higher we set the threshold to the more sets of keypoints will come back. But if the Levenshtein distance is too low it may miss the key points.

It is important to remember that the Levenshtein distance works as a holistic to find matching key points but it is very often the case that non matching images with have very close hashes. This is due to the nature of the hash not being design specifically for this purpose. In future work I would love to design my own hashing function that would do this better and give more accurate results. The more accurate the hash function can be the faster this application will become. This will make a huge difference to the speed because every set of keypoints that come from the query are matched against the original image to find the image with the highest amount of keypoints matching. The less of these that there are, will dramatically increase the speed of this search.

To get the most of speed out of the query, we want to calculate the Levenshtein distance as close to the database as possible. If we were to query the database and then run the Levenshtein distance in python this would be an almost worthless task. Instead we can implement plsql. This is an extreme learning curve close to the deadline and one that was not accounted for in the gantt chart at the planning phase of the project. It was therefore elected to use the function already built into the extra modules of PostgreSQL. This was a fast solution and one that worked very well.

Problems Solved

The problem that we set out to solve was the identification of complex object inside of images. With this we can hope to find images that are using copyrighted material inside images, or if watermarks, hues, and cropping are used we can still identify an object such as an artwork.

We have also solved the scalability problem of this image search. We have verified that it is a usable application can handle a number of users searching simultaneously.

Future work

Currently the locally sensitive hashing function is restricting the performance and is not similar enough for key points. The next version of this application would make use of better locally sensitive hashing function for this application, or be it a custom

hashing function. This would allow the database query to find less results and therefore speed up the KNN search on the key points.

The user interface is very basic. Increasing the accessibility of this user interface and making this interface accessible on mobile would need to be done in future iterations.

Function Usage

data_management.py

```
class DataObject:
    '''
    Class to represent a keypoint to write to database

    self.pt: (Float, Float)
    self.size: Float
    self.angle: Float
    self.octave: Int
    self.class_id: Int
    self.desc: Numpy Array
    '''

    def __init__(self, pt, size, angle, response, octave, class_id, desc):
        '''
        Initializes variables for function
        '''

    def __str__(self):
        '''
        Returns string representation for output to database
        '''

def pack_keypoints(keypoints, desc):
    '''
    This will take a list of opencv keypoints and turn them into a list of
    DataObjects definined above

    keypoints: List
    desc: Numpy Array

    Returns: List (of DataObjects)
    '''

def unpack_keypoints(kp_lst):
    '''
    Take list of DataObjects and converts back to list of opencv keypoints

    Returns: (List, Numpy Array)
```

```
    '''

def connect_postgres():
    '''
    Opens a connection to a postgres database

    Returns: Database connection object
    '''

def write_postgres(dhash, datapoint, url='Unknown'):
    '''
    Takes a hash of datapoint, datapoint and source url
    and writes that to the database

    dhash: String (Hash of datapoints)
    datapoint: String (Keypoints of an image)
    url: String (Source of image)

    Returns None
    '''

def query_postgres(dhash):
    '''
    Reads in a hash and queries database for a similar hash

    dhash: String (Hash of keypoints)

    Returns: String (Results of query)
    '''
```

img_manipulation.py

```
def get_keypoints(img):
    '''
    Takes an image and returns its keypoints and descriptors

    img: cv2.imread object

    returns: (matrix of key points, matrix of descriptors)
    '''

def get_match(desc1, desc2):
    '''
    Takes two matrices of descriptors about and image
    and uses a Flann based matcher to match these matrices

    desc1: matrix of descriptors
    desc2: matrix of descriptors

    returns: list of matching descriptors
```

```
    '''

def rotate_img(image):
    '''
    Will rotate a matrix corresponding to an image

    image: Matrix corresponding to an image

    returns: Rotated matrix corresponding to an image
    '''

def load_img(img):
    '''
    Takes a file name and loads that image into an opencv matrix

    img: String

    returns: opencv matrix
    '''

def match_images(org_img, comp_img):
    '''
    Will attempt to match two images

    img1: string corresponding to filename of image
    img2: string corresponding to filename of image

    results: String, Percentage accuracy
    '''
```

References

1. Semantic texton forests for image categorization and segmentation
2. An Image Similarity Acceleration Detection Algorithm Based on Sparse Coding
3. n-SIFT: -Dimensional Scale Invariant Feature Transform
4. Image similarity comparison
5. Semantic Texton Forests for Image Categorization and Segmentation
6. Introduction to SIFT