Web application for crack classification and segmentation on masonry surfaces

Domantas Dilys

Summer 2021

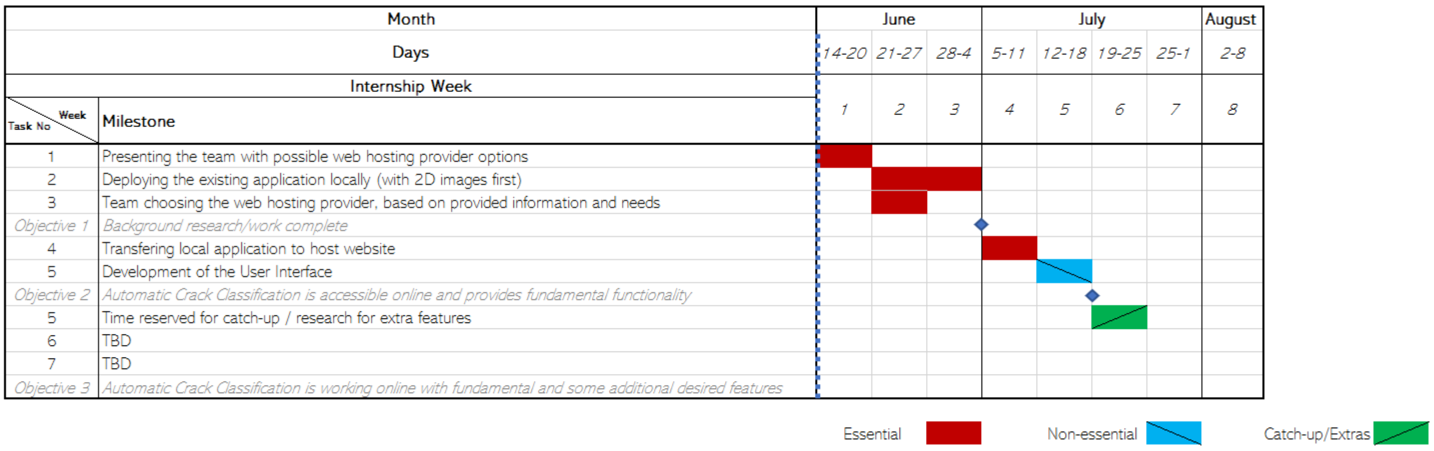
Contents

1 Goals of Web Application

The final aim of the web application is moving the wall crack detection algorithms that work locally to mobile platforms. However, in order to make the application easier to develop mobile applications on, the first good step is a web application, with potential future extensions for the mobile apps to make use of an API.

2 Plan of work

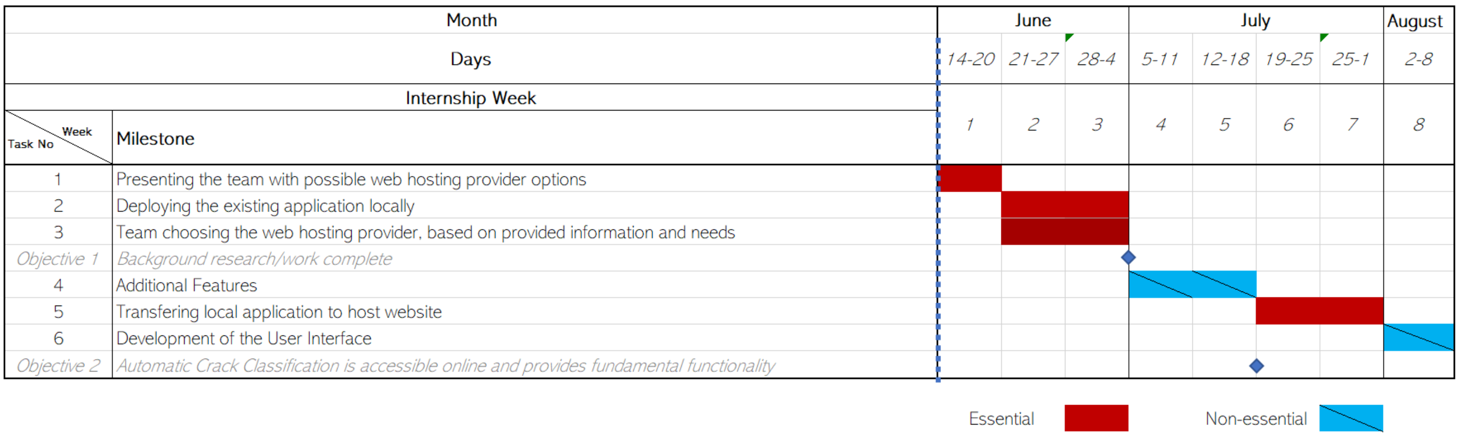
This chapter shows the planned tasks during the 8 weeks of the internship in summer of 2021. The tasks are shown in a Gantt-Chart in Figures 1 and 2



Month		June			July				August
Days		14-20	21-27	28-4	5-11	12-18	19-25	25-1	2-8
Internship Week									
Task No / Week	Milestone	1	2	3	4	5	6	7	8
1	Presenting the team with possible web hosting provider options	🟥							
2	Deploying the existing application locally (with 2D images first)		🟥	🟥					
3	Team choosing the web hosting provider, based on provided information and needs		🟥						
Objective 1	*Background research/work complete*			◆					
4	Transfering local application to host website				🟥				
5	Development of the User Interface					🟦			
Objective 2	*Automatic Crack Classification is accessible online and provides fundamental functionality*					◆			
5	Time reserved for catch-up / research for extra features						🟩		
6	TBD								
7	TBD								
Objective 3	*Automatic Crack Classification is working online with fundamental and some additional desired features*								

Essential	🟥	Non-essential	🟦	Catch-up/Extras	🟩

Figure 1: Initial project plan presented as a Gantt Chart



Month		June			July				August
Days		14-20	21-27	28-4	5-11	12-18	19-25	25-1	2-8
Internship Week									
Task No / Week	Milestone	1	2	3	4	5	6	7	8
1	Presenting the team with possible web hosting provider options	🟥							
2	Deploying the existing application locally		🟥	🟥					
3	Team choosing the web hosting provider, based on provided information and needs		🟥	🟥					
Objective 1	*Background research/work complete*			◆					
4	Additional Features				🟦	🟦			
5	Transfering local application to host website						🟥	🟥	
6	Development of the User Interface								🟦
Objective 2	*Automatic Crack Classification is accessible online and provides fundamental functionality*					◆			

Essential	🟥	Non-essential	🟦

Figure 2: Actual project timeline presented as a Gantt Chart

3 Nature of the system

Since the application will be online, it requires an access point on the web. In this section the overall system nature is described to help determining the best web hosting option for this particular application. Each subsection reviews questions that are key to choosing the best web-hosting option.
Next section (4.1 Web hosting choice) will expand on this section, further explaining possible hosting options, pricing, advantages and disadvantages of each.
Points made in this section will be used to back-up the decisions made in section 4.1.

3.1 Kind of Website

The website is planned to be a web application, containing interactive elements such as uplodad/download of images as input and output, logging in, running the machine learning classification algorithm and viewing the classified output.

3.2 Functionality

Customer database, with login option, storing images the users have uploaded to the system and the results, dates, and geo information.

3.3 Expected Traffic growth

The traffic growth should be quite steady, however, traffic spikes are possible for building inspections after earthquakes or the service being advertised.

3.4 Need of having multiple subdomains

Multiple websites and subdomains possible, as the team might like to extend showing their results in 3D, which could be viewed as a separate service (in addition to 2D image classification)

3.5 Expansion/Scaling-up

As indicated in subsection 3.4, the team is actively working in the field, therefore, any new functionality being added is likely.

3.6 Programming and Development requirements

Since the Machine Learning algorithm for crack classification in masonry is developed using Python, the language should be available, as well as frameworks such as Flask or Django. This set of available development tools would ensure smoother deployment of the application.

3.7 Impact of downtime

The application is not business-critical, therefore, engineers using the application can continue their work offline. Downtime is still a big inconvenience for engineers who would use it for building inspection and archiving, however, photos of the sites can be taken and uploaded when the service becomes available again.

4 Going Online

When the project needs to go online, some machine (server) needs to continuously run, thus supporting the application and providing access to the web. The choice of how and where the application will be deployed is an important one, as it shapes the design and capabilities of the web application in the future. Therefore, it is beneficial to consider long-term goals when making the initial choice, so that scalability does not become a concern. The application might not have big traffic or require a lot of storage initially, but if the service grows, migration costs might be significant later on.

This section describes key differences between web and cloud hosting and lists the potential providers based on the nature and goals of the application.

4.1 Web Hosting

This subsection summarises the three different options one might use: shared hosting, VPS (virtual private server) hosting and dedicated server hosting, as well as lists providers, where Python could be used, with potential advantages and disadvantages.

4.1.1 Shared hosting

This section briefly summarises how Shared hosting works, gives some providers that offer this service, as well as what limitations might be run into in the scope of this application

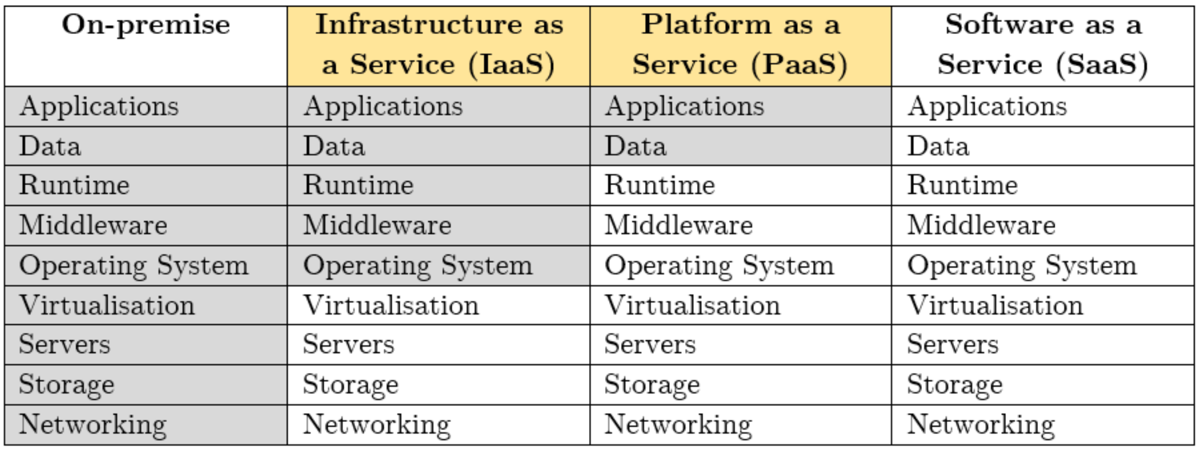
Shared web hosting works by having a single machine for many websites and the resources are allocated by usage. This has a potential limitation when websites on the same machine take up a lot of resources affecting performance of other websites. To avoid that, shared hosting providers might impose limits on the available resources. Furthermore, it is typical that the providers manage the machines, therefore, installing new software and packages is non-trivial. When choosing shared-hosting services, it is up to the user to do the research beforehand whether the software that they will need for the web application/web site are available.

4.1.2 Points to consider

- Web hosting providers that allow tiered services should be prioritised, since it makes scalability easily manageable.
- Unlimited data storage is often only offered at the high-end price range
- Unlimited bandwidth might be offered at the lower-end price range, however, if not, it must be made sure what the outcomes are when this bandwidth is exceeded when a sudden traffic spike occurs (might result in an outage, or an additional payment)
- Some providers might limit number of pages on the website, therefore, it is good to know before-hand how many shall be needed
- Uptime guarantees should be close to 99%, otherwise, problems might arise in the long run.
- Security is of high importance, therefore, the web service should allow manual or automatic backup of the site. Furthermore, Denial of Serive (DDoS) protection should be mentioned and provided.

4.2 Cloud Hosting

An alternative to web hosting are cloud hosting services. This is quite a new approach in the world of web applications. Since the application is to be developed ourselves, the services that can be considered are infrastructure and platform (highlighted yellow in Figure 3. These will be discussed next, with their advantages, disadvantages given the nature of the application to be deployed, as well as a list of possible providers.



On-premise	Infrastructure as a Service (IaaS)	Platform as a Service (PaaS)	Software as a Service (SaaS)
Applications	Applications	Applications	Applications
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
Operating System	Operating System	Operating System	Operating System
Virtualisation	Virtualisation	Virtualisation	Virtualisation
Servers	Servers	Servers	Servers
Storage	Storage	Storage	Storage
Networking	Networking	Networking	Networking

Figure 3: This section discusses Infrastructure and Platform as a Service. Discussion about Software as a Service is emitted since an application is to be deployed.

4.2.1 Infrastructure as a Service (IaaS)

Instead of managing every aspect of the project on-premises, from scratch, infrastructure can be provided. This includes Networking, Storage, Servers and Virtualisation (virtualisation is a concept where each server hosts numerous virtual machines, made available to clients). This infrastructure listed before is now maintained and managed by the provider, leaving the client to manage the OS, Middleware, Runtime, Data and the Application.

An Example of such service providers are given in the table below.

4.2.2 Platform as a service (PaaS)

What differentiates Infrastructure and Platform Services, is the amount of setting-up you are expected to do. In the Platform, most things are already set-up, and those Platforms are typically prices according to the underlying resources offered. This means, it is typically the case that switching a Platform might require only to tweak some configuration files, as each offered platform already has the underlying services set-up.

4.3 Comparison of the hosting options

The Choice of the five identified options is given in Appendix

5 Appplication

This chapter is a documentation for the application that was developed in the 5 weeks during the 2021 summer internship. The key sections describe the Structure and how to run and deploy the application locally, as well as on the cloud service.

5.1 Application Structure

This implementation follows the usual Django/Flask file structure, where one can find URL mappings in `urls.py` and all the HTML templates in the directory `mcd/templates/mcd/*.html`.

The key files are:
1. `settings.py` - this has the database settings, media routes, paths etc.

2. `urls.py` - there are two, at `mcd/urls.py` and `mcd_webapp/urls.py` (root/mcd/... urls are handled in `mcd/urls.py` and root/... urls - in `mcd_webapp/urls.py`)

These URL files give the mappings - what URL should call which python function. The python functions associated in each given URL can be found in `mcd/views.py`.

3. `main.py` - to work with Google Cloud, the file was modified from the default Django to:

```
from mcd_webapp.wsgi import application

# App Engine by default looks for a main.py file at the root of the app
# directory with a WSGI-compatible object called app.
# This file imports the WSGI-compatible object of your Django app,
# application from mysite/wsgi.py and renames it app so it is discoverable by
# App Engine without additional configuration.
# Alternatively, you can add a custom entrypoint field in your app.yaml:
# entrypoint: gunicorn -b :$PORT mysite.wsgi
app = application

# Configure this environment variable via app.yaml
import os
CLOUD_STORAGE_BUCKET = os.environ['CLOUD_STORAGE_BUCKET']
```

4. `mcd_webapp/__init__.py` - to work on Google Cloud, this file is set to:

```
# main application
import pymysql

pymysql.install_as_MySQLdb()
```

5. `app.yaml` - this is the configuration file that Google Cloud reads when deploying! It has all the options, to learn more about `yaml` files, please read here:
`https://cloud.google.com/appengine/docs/standard/python3/config/appref`

```
# [START django_app]
runtime: python38

handlers:
# This configures Google App Engine to serve the files in the app's static
# directory.
```

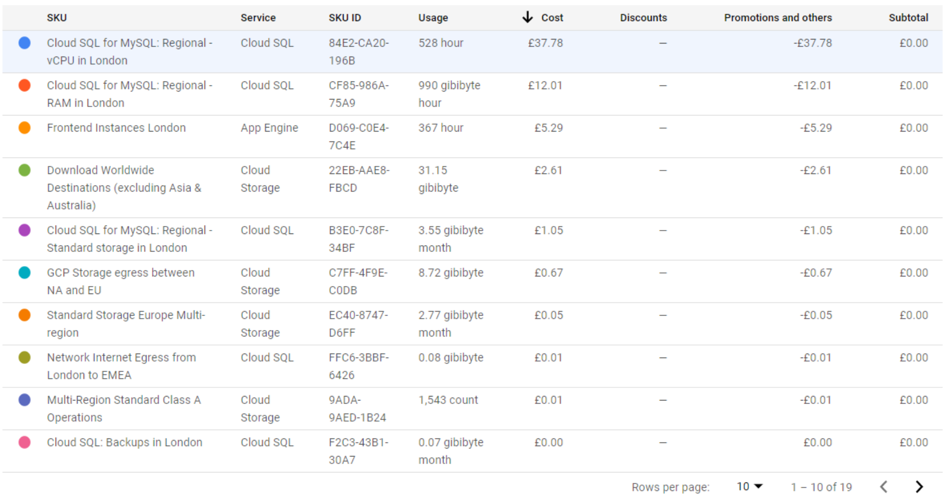
```
- url: /static
  static_dir: static/

# This handler routes all requests not caught above to your main app. It is
# required when static routes are defined, but can be omitted (along with
# the entire handlers section) when there are no static files defined.
- url: /.*
  script: auto
# [END django_app]

env_variables:
    CLOUD_STORAGE_BUCKET: mcd_file_storage

instance_class:
  F4_1G
automatic_scaling:
  max_instances: 1 (--> you can adjust this as you wish)
  min_instances: 0 (--> will scale to 0 when there is no traffic so won't incur costs)
```

Here you can see, that Python version 3.8 is used, and the `instance_class` is F4_1G, you can see information about the instance classes here:
`https://cloud.google.com/appengine/docs/standard#second-gen-runtimes`

Automatic Scaling here makes sure that at maximum, one instance is running, and if there is no usage - 0 instances run, meaning that Google App Engine should not cost anything for that time it is not running.

5.2 Disabling the Application

Since the costs can grow if untracked (Figure 4 shows the first month cost distribution), it is good to disable the Application when unused. It will be shown here how to do it.



	SKU	Service	SKU ID	Usage	↓ Cost	Discounts	Promotions and others	Subtotal
●	Cloud SQL for MySQL: Regional - vCPU in London	Cloud SQL	84E2-CA20-196B	528 hour	£37.78	–	-£37.78	£0.00
●	Cloud SQL for MySQL: Regional - RAM in London	Cloud SQL	CF85-986A-75A9	990 gibibyte hour	£12.01	–	-£12.01	£0.00
●	Frontend Instances London	App Engine	D069-C0E4-7C4E	367 hour	£5.29	–	-£5.29	£0.00
●	Download Worldwide Destinations (excluding Asia & Australia)	Cloud Storage	22EB-AAE8-FBCD	31.15 gibibyte	£2.61	–	-£2.61	£0.00
●	Cloud SQL for MySQL: Regional - Standard storage in London	Cloud SQL	B3E0-7C8F-34BF	3.55 gibibyte month	£1.05	–	-£1.05	£0.00
●	GCP Storage egress between NA and EU	Cloud Storage	C7FF-4F9E-C0DB	8.72 gibibyte	£0.67	–	-£0.67	£0.00
●	Standard Storage Europe Multi-region	Cloud Storage	EC40-8747-D6FF	2.77 gibibyte month	£0.05	–	-£0.05	£0.00
●	Network Internet Egress from London to EMEA	Cloud SQL	FFC6-3BBF-6426	0.08 gibibyte	£0.01	–	-£0.01	£0.00
●	Multi-Region Standard Class A Operations	Cloud Storage	9ADA-9AED-1B24	1,543 count	£0.01	–	-£0.01	£0.00
●	Cloud SQL: Backups in London	Cloud SQL	F2C3-43B1-30A7	0.07 gibibyte month	£0.00	–	£0.00	£0.00

Rows per page: 10 1 – 10 of 19

Figure 4: Cost distribution of the first month, SQL Instances have been the biggest cost (was using 2 virtual CPUs, SSD 10GB Storage, multi-regional and with auto-backups. After August 12th 2021, settings were chosen to only use 1 virtual CPU, HDD cheaper 10GB Storage, only single region support and no auto-backups, so the price should decrease for future months

This Google Project consists of three main Google Services:
1. Google App Engine - `https://console.cloud.google.com/appengine`
2. Google SQL Instance - `https://console.cloud.google.com/sql/instances/polls-instance/`
3. Google Cloud Storage - `https://console.cloud.google.com/storage/browser`

So, in order to fully stop the application, the App Engine and the SQL Instance can be disabled/stopped. As mentioned above in the `app.yaml` file we set that if the service is idle, it uses 0 instances anyway. Therefore, costs for Google App Engine just sitting there should not be high anyway.

5.2.1 Stopping Google SQL Instance

What we can see from 4, the most expensive was the SQL Instance storage. After 12th of August, the settings were changed to bare minimum so that if left running it should not exceed 30£ in future months, however, it is still desirable to know how to disable it:
Use this link:
`https://console.cloud.google.com/sql/instances/polls-instance/overview?project=cool-keel-320414`
and click 'Stop', it will remain stopped until you start it again manually.
It might take up to 5 minutes to stop. To confirm it has stopped, go to the mcd application link and you should see this error:
`"Can't connect to MySQL server on 'localhost' ([Errno 111] Connection refused)")`

5.2.2 Stopping Google App Engine

As mentioned before, the App Engine is now set to the mode, where it will barely use any resources if noone is connecting to it (since `min_instances: 0` set in `app.yaml`, and it only cost 5.29£ in the first month, so in the future it would be even less. However, to really turn it off, there are two ways:

1. You can always delete ALL versions here if needed:
`https://console.cloud.google.com/appengine/` (but the downside is that you will have to then redeploy with `gcloud app deploy`, so I suggest first trying the 2nd way below, and then see how much the costs go down - they should be close to 0.00£ with the second option, but in case not - try this first option)

2. [Recommended First] You can disable the application at the link:
`https://console.cloud.google.com/appengine/settings`
and click 'Disable application' blue button. (just make sure you are logged in with an account authorised to access the project `cool-keel-320414` (which is the automatically generated name for this project). To confirm it stopped, you should see error `The requested URL was not found on this server.` when you go to the application link.

Another way to disable the application is this:
`https://stackoverflow.com/questions/27201832/how-to-stop-or-disable-google-app-engine-production-server`
(you can use this if you have disabled it before, but for some reason the costs are still not 0.00£ for the Google App Engine.

5.2.3 Stopping Google Cloud Storage

I have not found a way to 'disable' Cloud Storage, because it just stores files, there is no associated CPU with it for the user. However, it only cost 0.67£, so it does not seem a problem.
The only way is to actually delete the Bucket:
`https://console.cloud.google.com/storage/browser`
but that means any time you want to run the application on the cloud again, you would have to set it all up all again, and even change one setting in the code to handle the new Google Bucket name, so if the cost stays around the same, I would not bother deleting it for now.

5.3 Running the Application

This subsection describes how to run the program locally and deploy it on the cloud.

5.3.1 Running Locally

Key point to note here is that when developing locally, the application database is still connected to the cloud. This can be achieved by using Google Cloud SDK.
Next, a step-by-step guide will help to run the application locally.

1. Using an IDE such as PyCharm might make the process much easier, however, this guide will help to run the application using the command-line terminal so that any IDE can then be used to edit the code.

Navigate to where you want the project to be located on your machine.
In your command line, run:
`git clone https://github.com/MCDservice/mcd_webapp.git`
NOTE: repository name is mcd_webapp, make sure the '_' (underscore) symbol is there when you copy the cloning

URL! Otherwise you will get an error saying that 'mcdwebapp repository does not exist

2. Once the repository has been copied, navigate to the `mcd_webapp` folder, where the `main.py` and `manage.py` files are.
Then, the next steps follow the steps given at Google Cloud - Running Django on Google App Engine:
`https://cloud.google.com/python/django/appengine` from Step 2 onwards.

```
python -m venv env
(on Windows, see the link above for commands on Mac/Linux):
venv\scripts\activate
pip install -upgrade pip
pip install -r requirements.txt
```

3. In the meantime (while pip is installing requirements, or if it has done so), install Google SDK.
First, install the sql proxy from here:
`https://cloud.google.com/sql/docs/mysql/sql-proxy#install`
and put the file `cloud_sql_proxy_x64.exe` to the same directory (`mcd_webapp`), where the `main.py` and `manage.py` files are. AND rename the file to just:
`cloud_sql_proxy.exe`

Then, follow the intructions here to install Google SDK:
`https://cloud.google.com/sdk/docs/install`

4. When Google SDK is installed and `cloud_sql_proxy.exe` is in the correct location,
run the Google Console (application called `Google Cloud SDK Shell`, you can find it using Windows Search or App Seach for example)

5. In the Google SDK Shell, navigate to where you placed the `cloud_sql_proxy.exe` file and run the following command:
(again, making sure the underscores '_' are copied properly when copying this command above)

```
cloud_sql_proxy.exe -instances="cool-keel-320414:europe-west2:polls-instance"=tcp:3306
```

(the command above works on Windows, for other OS and overall more general instructions please see:
`https://cloud.google.com/python/django/appengine#run-locally`)

NOTE: It might ask you to authenticate, then you will be redirected to log into a Google Account, which has to be one of the accounts that are Owners/Editors of the project. If your account does not have the rights, please ask the administrator to add you via the link:
`https://console.cloud.google.com/iam-admin/iam?serviceId=defaultcloudshell=trueproject=cool-keel-320414`

If everything worked, this should then give you the following message:

```
Listening on 127.0.0.1:3306 for cool-keel-320414:europe-west2:polls-instance
```

6. Then, in your other console, where `pip` was installing requirements (it should now be finished), run:
`python manage.py runserver`
This **should** give you an ERROR:

```
FileNotFoundError:  [Errno 2] No such file or directory:  'mcd_webapp/key1.txt'
```

This is simply an error because the key.txt files are passwords, which are not stored in the repository for anyone to publically clone them!
Therefore, you must create files (in the same folder as `mcd_webapp\mcd_webapp\getKey.py`):
a) `key1.txt`, containing password of `/cloudsql/cool-keel-320414:europe-west2:polls-instance` and nothing else,
b) `key2.txt`, containing password of `mcd.recover@gmail.com` and nothing else.
Also important, check if `RUN_LOCALLY = True` in `views.py` and that you have placed the DefaultCredentials .json file where `manage.py` is. Otherwise, views.py comment gives details how to deal with `DefaultCredentialsError`

7. Run again:
`python manage.py runserver`
This should now not give errors. IF it does, and it is something like:

```
django.db.utils.OperationalError:  (1045, "Access denied for user 'polls'@'cloudsqlproxy 81.100.235.69'
(using password:  YES)")
```

This means the password you gave in `key1.txt` is incorrect. IMPORTANT: check that the file DOES NOT contain and end-line character at the end! It should just be LINE 1 length with no LINE 2.
If there are 2 lines in the key1 and key2 textfiles, delete the excess lines, save the files, exit the error message above with CTRL+C, and run `python manage.py runserver` again.

If it works, you should get message:
`Starting development server at http://127.0.0.1:8000/ Quit the server with CTRL-BREAK.`
CTRL-BREAK is usually CTRL+C, to stop the server running! It should re-run automatically as files are updated, and if you want to make sure it runs again, just type `python manage.py runserver` as usual.

5.3.2 Deploying on the Cloud

1. Having logged in with mcd.ml.service@gmail.com, Navigate to:
`https://console.cloud.google.com/appengine?authuser=4project=cool-keel-320414serviceId=default`
And turn on the Cloud Shell (shown in Figure 7)

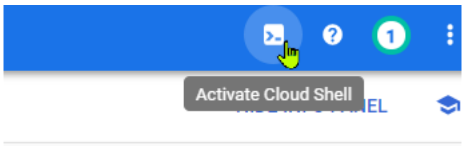


Figure 5: How to turn on the Cloud Shell (top right of the menu, in 2021)

2. In the cloud shell, if you have logged in for the first time, it might ask you to add the project:

`To set your Cloud Platform project in this session use “gcloud config set project [PROJECT_ID]”`
To which you should just respond with:
`gcloud config set project cool-keel-320414`

3. If this is the first time you logged on here, the project will not be there. You need to clone it from a git repository, just like you did locally:
`git clone https://github.com/MCDservice/mcd_webapp.git`
(again, please make sure underscores '_' are included if you copied the line above)

If you have logged on before, then you need to run: `git pull`
Before running activate virtual environment using: `source venv/bin/activate`

This will get any changes you've made to the project. Using `git` is the easiest way to get your changes visible in the cloud.

4. The following instructions will basically follow Google's advice, found here:
`https://cloud.google.com/python/django/appengine`
But for the sake of convenience, we list the steps to take here too:

a) if you made changes to the `styles.css` file, run:
`python manage.py collectstatic`
This will move the style files to the correct location so that the web application has access to them.
If you have done that and styles still don't work, it might be that styles are cached! In that case, a quick fix is CTRL+F5 (hard reset).

b) finally, run:
`gcloud app deploy` (or use `gcloud app deploy -version v1` if a previous version eists). And remember to delete the old versions here:
`https://console.cloud.google.com/appengine/versions?project=cool-keel-320414`
It will take some time, but in the end it should give a message as such:
`To view your application in the web browser run:  gcloud app browse`
And that's it! You can access the application given the current (2021 August) link:
`https://cool-keel-320414.nw.r.appspot.com/mcd/`.

6 Longer-term Maintenance

This section gives the key links of Google Cloud, and gives instructions how to view the Database and set its CORS options.

6.1 Domain Name and CORS

The current object database is set to have the following CORS (Cross-origin Resource Sharing) enabled:

```
["maxAgeSeconds": 3600,
"method": ["GET"],
"origin": ["http://127.0.0.1:8000", "https://cool-keel-320414.nw.r.appspot.com"],
"responseHeader": ["Access-Control-Allow-Origin"]]
```

(you can view this in the same console, that can be found in URL:
`https://console.cloud.google.com/appengine?project=cool-keel-320414serviceId=defaultcloudshell=true`
and by running the command:
`gsutil cors get gs://mcd_file_storage`)

But if a new domain name is added, then this `cors.json` file should be updated to allow the new domain:

```
"origin": ["http://127.0.0.1:8000",
"https://cool-keel-320414.nw.r.appspot.com",
"https://mcdnewdomain.com"]
```

Therefore, to update the CORS, you should do the following:
if `cors.json` file is NOT in the current folder in the cloud terminal (you can check by typing `ls`, but it should be placed in
codemcd_webapp/, where `manage.py` is, for example), then you can type `touch cors.json`, and this will make an empty file.
Open that file with `nano cors.json` and copy the old settings:

```
[
    {
      "origin": ["http://127.0.0.1:8000", "https://cool-keel-320414.nw.r.appspot.com"],
      "method": ["GET"],
      "responseHeader": ["Access-Control-Allow-Origin"],
      "maxAgeSeconds": 3600
    }
]
```

and change it to something like:

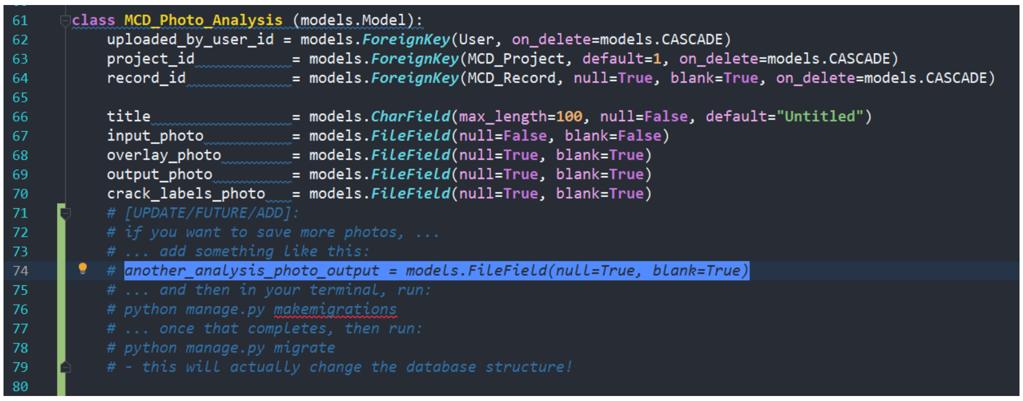
```
[
    {
      "origin": ["http://127.0.0.1:8000", "https://cool-keel-320414.nw.r.appspot.com", "https://
          newdomain.com],
      "method": ["GET"],
      "responseHeader": ["Access-Control-Allow-Origin"],
      "maxAgeSeconds": 3600
    }
]
```

Then, save the file, and set the cors with:
`gsutil cors set cors.json gs://mcd_file_storage`

The CORS is used on the Progress Bar, when the user uploads an image and we query the status from the database, therefore, to check if the CORS worked with the new domain, you can test if the progress-bar is working and look at the Console output (CTRL+SHIFT+I and then go to Console, in the inspection mode)

6.2 Adding New Analysis to Database

If you want more outputs from current model, or simply want to add another model of analysis, you will first need to add a new entry in the database as such:



```
class MCD_Photo_Analysis (models.Model):
    uploaded_by_user_id = models.ForeignKey(User, on_delete=models.CASCADE)
    project_id          = models.ForeignKey(MCD_Project, default=1, on_delete=models.CASCADE)
    record_id           = models.ForeignKey(MCD_Record, null=True, blank=True, on_delete=models.CASCADE)

    title               = models.CharField(max_length=100, null=False, default="Untitled")
    input_photo         = models.FileField(null=False, blank=False)
    overlay_photo       = models.FileField(null=True, blank=True)
    output_photo        = models.FileField(null=True, blank=True)
    crack_labels_photo  = models.FileField(null=True, blank=True)
    # [UPDATE/FUTURE/ADD]:
    # if you want to save more photos, ...
    # ... add something like this:
    # another_analysis_photo_output = models.FileField(null=True, blank=True)
    # ... and then in your terminal, run:
    # python manage.py makemigrations
    # ... once that completes, then run:
    # python manage.py migrate
    # - this will actually change the database structure!
```

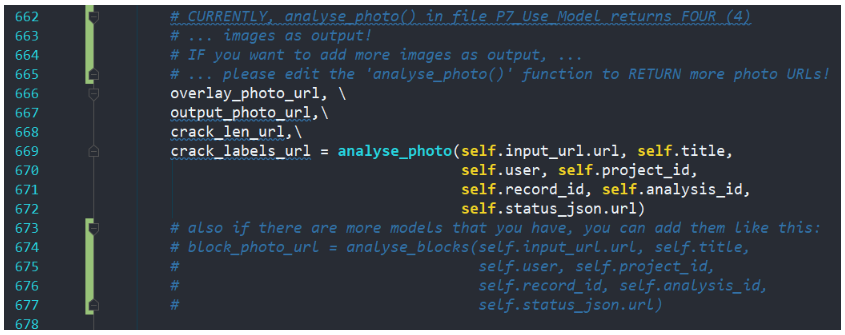
Figure 6: Add new line in models.py for every new image you want to save

Then it does not end there. You now need to migrate the changes, which Django makes quite simple:
In your terminal, just type:
`python manage.py makemigrations`
and
`python manage.py migrate`

Then comes the harder part - connecting the application.
Currently, in `P7_Use_Model.py` function `analyse_photo()`, it returns four URLs like this:

```
    return url_dict["Overlay"], \
           url_dict["Binarised (t=" + str(Bin_Threshold) + ")"], \
           url_dict["crack_len_csv"], \
           url_dict["Final Watershed"]
```

if you have a new `Model_Use` file, you will need to do something similar.
Then, in `views.py` class `EnqueuePhotoAnalysis`, you need tocall this new model, and save the new URLs to database:
Then, you have to actually save those new URL paths to database::



```
        # CURRENTLY, analyse_photo() in file P7_Use_Model returns FOUR (4)
        # ... images as output!
        # IF you want to add more images as output, ...
        # ... please edit the 'analyse_photo()' function to RETURN more photo URLs!
        overlay_photo_url, \
        output_photo_url,\
        crack_len_url,\
        crack_labels_url = analyse_photo(self.input_url.url, self.title,
                                         self.user, self.project_id,
                                         self.record_id, self.analysis_id,
                                         self.status_json.url)
        # also if there are more models that you have, you can add them like this:
        # block_photo_url = analyse_blocks(self.input_url.url, self.title,
        #                                   self.user, self.project_id,
        #                                   self.record_id, self.analysis_id,
        #                                   self.status_json.url)
```

Figure 7: Add new line in `EnqueuePhotoAnalysis` for every new image you want to save to database

```
    # after the photo has been analysed ...
    # ... put the links URLs of the output images on the cloud ...
    # ... in our User Database!
    t.output_photo        = output_photo_url    # change field
    t.overlay_photo       = overlay_photo_url   # change field
    t.crack_labels_csv  = crack_len_url         # change field
    t.crack_labels_photo = crack_labels_url     # change field
    t.analysis_complete  = True                 # change field

    # ADD NEW!
    # if you add MORE outputs, add something like:
    # t.another_analysis_output = another_analysis_output_URL # change field
```

And last, please update the delete functions:
search in the `views.py` (CTRL+F) the usage of `delete_file_from_cloud_media`, and add new line in each section as such:

```
            # if you added more files as output ...
            # ... please add more cleanup, like this:
            # delete_file_from_cloud_media(some_another_file)
```

There are THREE Sections in `views.py` where you will need to update this:
1. in `EnqueuePhotoAnalysis`, where `delete_file_from_cloud_media` is called if the analysis fails;
2. in `delete_record(record_to_delete)`;
3. in `PhotoAnalysisDelete(DeleteView)`.

6.3 Key Links

These are the key links used in production:

1. Project Homepage:
`https://console.cloud.google.com/home/dashboard?project=cool-keel-320414`

2. Disabling/Turning Off Project - Cost Management:
`https://console.cloud.google.com/appengine/settings`

3. Project Versions:
`https://console.cloud.google.com/appengine/versions?project=cool-keel-320414`
Delete old versions, because otherwise you would have to pay extra for keeping them there! (or use `gcloud app deploy -version v1` instead of `gcloud app deploy`)

4. Project Running Logs:
`https://console.cloud.google.com/logs/`
Extremely useful when debugging live website interactions, as it might display errors for why the analysis got stuck, for example, the "Out of Memory" errors.

5. Project Database (MySQL, where the Users are stored and links to images etc.):
`https://console.cloud.google.com/sql/instances/polls-instance/overview?project=cool-keel-320414`

6. Project Bucket (where Images, Model etc. are stored):
`https://console.cloud.google.com/storage/browser/mcd_file_storage`
The MODEL itself is stored here:
`https://console.cloud.google.com/storage/browser/mcd_file_storage/pretrained-model/model-for-crack-detection`

The configuration app.yaml file:
`https://cloud.google.com/appengine/docs/standard/python3/config/appref`

Read more about instance classes of app.yaml Google Cloud:
`https://cloud.google.com/appengine/docs/standard#second-gen-runtimes`

Convenient Documentation:

See how people manage Billing and Costs to avoid unexpected surprises (people sharing bad experiences and how to avoid them):
`https://stackoverflow.com/questions/47125661/pricing-of-google-app-engine-flexible-env-a-500-lesson`

How to View and Edit Logs:
`https://cloud.google.com/appengine/docs/standard/python3/writing-application-logs`

Running Django on AppEngine:
`https://cloud.google.com/python/django/appengine`

Reading and Writing from/to Cloud Storage:
`https://cloud.google.com/appengine/docs/standard/python3/using-cloud-storage`

Quotas and Running Usage:
`https://console.cloud.google.com/iam-admin/quotas?orgonly=trueproject=cool-keel-320414`

Creating Google Cloud Instances:
`https://cloud.google.com/sql/docs/mysql/create-instance`

Appendix