Master Project Report

Platform for Bee Tracking in Videos
11.12.2020

Anna Jancso

Vladimir Masarik

1 Intro

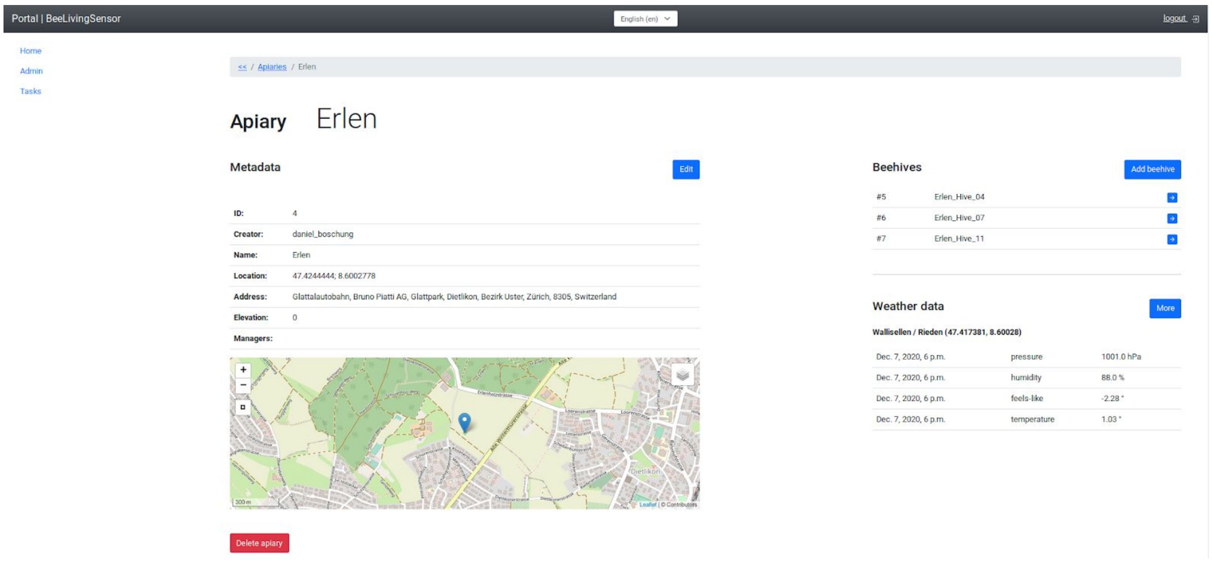
BeeLivingSensor (BLS) is a joint project between the Institute of Molecular Systems Biology (ETH) and the Informatics and Sustainability Research Group (UZH). It aims to understand how various factors such as the weather, agronomy, landscape and diseases influence the life and well-being of honeybee colonies in a natural, not industrialized setting. The main goal is to build an open, web-based platform that aggregates data from different sources in an non-invasive way. The core source of data are the video-surveillance cameras that capture the movements, interactions and the pollen input of honeybees. This video-data is enriched with data from the beehive (temperature, humidity, weight, sound, etc.) and the environment (weather data, drones images, agricultural data, etc.). The platform is designed as a citizen science project where different stakeholders including beekeepers, farmers, schools, etc. help collect data that researchers may use for analysis.

The project started back in the summer of 2019 where a first group of students at ETH built an AI model to detect bees in videos. Another group from UZH continued this work in February 2020 who refined that model. We started our master project in July 2020 for which we created the first version of the platform that integrates the AI models built by the other groups. Apart from bee detection in videos uploaded by beekeepers, the platform currently offers features such as user and group management, apiary and beehive management, task-based image labeling, model validation and training as well as aggregation of sensor and weather data and their visualization in graphs. On the non-functional side, the platform was built with scalability and availability in mind by leveraging the powers of Kubernetes. Moreover, it relies on common security standards such as HTTPS and view access permissions. We also set up a CI/CD pipeline that runs automated tests for each commit and allows deployment of the Kubernetes cluster at the push of a button.

In October 2020, the project received a grant from Microsoft for a sub-project called the *BeePollenTracker* that aims to recognize the amount and color of pollen coming into the beehive which is a key marker of biodiversity. This funding enabled us to use all the necessary computational resources (virtual machines, storage, etc.) to host our platform for free which is now fully operable under https://beelivingsensor.eu, a screenshot of it shown in Figure 1.

In the future, the platform will be extended to include AI models for the pollen detection that are currently under development by another ETH group and correlate it with plant data. Furthermore, the BLS team also envisages to incorporate agricultural data and drones/satellites data, and train AI models for detecting the Varroa disease in bees. Since the platform has only been online for a couple of weeks and has only been available to admins so far, another large milestone will be to test the platform with ‘real’ users to determine the platform’s user-(un)friendliness and performance.

In the following, we will document our work in more detail and how the platform can be set up locally and in production.



Apiary Erlen

Metadata

ID:	4
Creator:	daniel_boschung
Name:	Erlen
Location:	47.4244444; 8.6002778
Address:	Glattalautobahn, Bruno Piatti AG, Glattpark, Dietlikon, Bezirk Uster, Zürich, 8305, Switzerland
Elevation:	0
Managers:	

Beehives

#5	Erlen_Hive_04
#6	Erlen_Hive_07
#7	Erlen_Hive_11

Weather data

Wallisellen / Rieden (47.417381, 8.60028)

Dec. 7, 2020, 6 p.m.	pressure	1001.0 hPa
Dec. 7, 2020, 6 p.m.	humidity	88.0 %
Dec. 7, 2020, 6 p.m.	feels-like	-2.28 °
Dec. 7, 2020, 6 p.m.	temperature	1.03 °

Figure 1: Detail view of an apiary.

2 Platform overview

In this section, we describe functional and non-functional requirements that we have implemented in the project. We outline the platform’s high-level architecture explaining every key component, describe our repository’s structure and provide instructions for running the platform both locally and in production.

2.1 Functional

We structure the functions in sections and will explain them in a narrative way, i.e. following the natural flow of steps that users take when they navigate through the platform.

User/Group management

As users access the platform for the first time, they are presented with a login page. Users that have no account yet have to register first by clicking on the register link. Once the registration process is finished, users can sign in and are redirected to the Home page. At this point, new users have no permissions whatsoever and have to wait for the admins to give them appropriate roles that are associated with given permissions. Currently, eight roles (called ‘groups’) are available, e.g. *beekeeper* or *researcher*. The admins can add and remove roles, change their associated permissions and assign them to users.

Apiary/Beehive management

Assuming the user receives beekeeper status, (s)he can add apiaries, supplying their names and locations. The location is used to automatically retrieve weather measurements of the closest weather station. The beekeeper can go on adding beehives for an apiary. As new beehives have no AI model for bee detection set yet, the beekeeper has to select the most similar beehive based on images that the admin provides.

Incorporation of sensor data

Upon creation of a beehive, beekeepers may add sensor devices. For each sensor device, sensor types can be added. Currently, there are six sensor types available such as temperature or weight. The admins can define which sensor types are available and also add new ones in the admin page. Once a sensor has been added, measurements can be manually added. As of now, we only provide a half-automated approach for adding sensor measurements, namely for CAPAZ[1] devices. Since CAPAZ does not provide an API, beekeepers have to export the measurements on the CAPAZ website as an Excel spreadsheet. This spreadsheet can be uploaded on the platform which handles its processing and automatically adds sensors and their measurements contained within it.

Video Upload / Annotation

After creating a beehive, the beekeeper may upload videos of it. The platform processes these videos to extract their frames and detect bees in each frame. For this, it uses the three best AI models of the most similar beehive that the beekeeper selected when creating the beehive. The platform randomly chooses one frame and presents the annotations of the three models to the beekeeper who decides which model created the most accurate annotations.

Frame Labeling

To find an AI model that creates precise annotations for the new beehive, the available AI models in the database are validated on it. This validation requires that labeled images of the beehive be available which is not the case for a new beehive. Therefore, when the beekeeper has uploaded a reasonable number of representative videos (currently three), a labeling task is automatically opened. These images can be labeled by anyone who has labeling permissions. Currently, only tasks for bee labeling are automatically opened. Tasks for other annotation types such as pollen or varroa labeling are not automatically opened. Although the admin can open such tasks manually and also add further annotations types, this has not been tested explicitly.

Model validation

Once each frame has been labeled by at least two annotators, the beehive is validated on each AI model that achieves a f1-score greater than 0.9 on any beehive. If a model is found that achieves a f1-score >0.9 for the new beehive, this model will be set for the beehive and

[1] https://www.capaz.de/stockwaage

no training is required. From that point on, videos of the beehive uploaded by the beekeeper are analyzed by that model. All performance metrics of each validation are recorded and can be viewed on the platform by the admins. Furthermore, the admin can see for which beehives a training is recommended.

Training

The training of new models is not fully automated in the sense that the admins cannot click on a button to start the training on the platform. Rather, the admins have to spin up the virtual machines on the Azure portal manually and execute a script in them.

Visualization of data

Once sensor measurements are added, these are visualized along with weather measurements on the beehive detail page in a graph. Currently, only a subset of the hive/weather parameters are shown on the graph.

2.2 Non-functional

Continuous integration & Continuous delivery

At the beginning of the project we have constructed a pipeline responsible for automated testing and deployment of the committed code. This helps us in finding out bugs, crashes, and other types of unintentional mistakes within minutes of pushing the code into our central repository. Additionally, this functionality also allows us to automatically apply the changes made into the production environment in an automated fashion, so that we are able to focus on development.

The continuous integration and continuous delivery (CI/CD) is implemented using GitLab, and the special file `.gitlab-ci.yaml`[2]. The file defines all necessary variables and has three jobs. First job is the test execution in the `unittest-webapp` section, which executes the tests written for the web application module, i.e. the web site. The second job is also test execution in the `unittest-modelservingapp` section, which executes tests for the model serving application, i.e. the module responsible for validating machine learning models. The last stage, which will be run only if there were no problems detected in the previous two stages, packages and prepares the updated code for deployment into the production environment.

The first two stages are the continuous integration, and the last one is the continuous delivery. Any variables that are used during these three stages and need to stay secret, such as passwords, private keys, are defined and stored using the GitLab web interface.

Internationalization

We also added support for multilingual pages following the standard approach in Django which relies on the GNU utility *gettext* that cleanly separates programming and translating. At the moment, the platform is offered in English and German. While all content is translated in English, only parts of the platform have been translated into German. Thus, localization is still lacking in some places, but the underlying support for it is there.

[2] https://docs.gitlab.com/ee/ci/yaml/

Security

Our platform is fully encrypted using the HTTPS. Moreover, we leverage all of Django's built-in security functions[3]. Furthermore, we also used Django's group-based permission system to restrict views to certain groups of people. To ensure that only authenticated users can access the views (except for the login and register pages), we also wrote a unit test that checks whether an URL requires a login. Currently, only three URLs (out of 80) are unprotected which we could not secure in time and for which we also have an open issue on GitLab.

2.3 Architecture

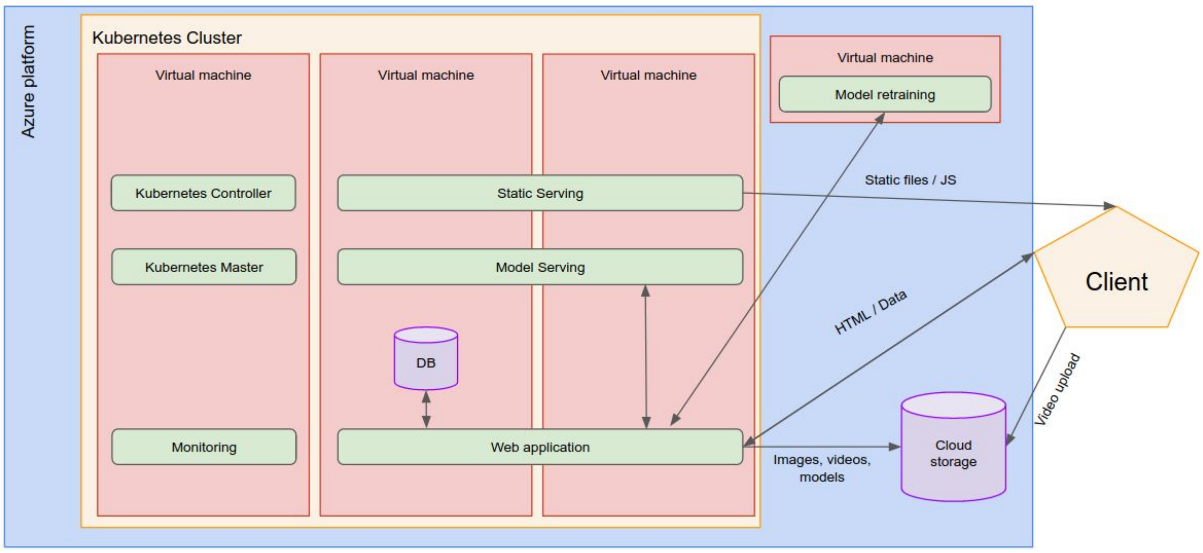


Figure 2: Architecture of our platform.

The platform consists of the following components:

- **Web application**: represents the front- and backend and therefore acts as the main interaction point for the client. We used Django for implementing the webapp which is a backend framework, i.e. routing and template creation is performed on the backend. Most of the platform's functionality like apiary, hive and user management, data visualization as well as the frame labeling are implemented there.
- **Database**: On the back-end, we run a PostgreSQL database extended with PostGis functionalities as our platform also collects geographic data such as coordinates of apiaries or weather stations. The database stores all non-binary data such as apiary and beehive metadata, user data and metadata of the videos, images and models.
- **Static serving**: serves all static files for the webapp such as the CSS and Javascript files. This allows us to improve the performance of loading the website because it takes special responsibility from the Web-app.
- **Model Serving application**: a REST API for the model serving. It exposes two endpoints, one for whole videos and the other for individual frames. In both cases, the frames (as extracted from the video) are run through an AI Model which detects bees in them. We implemented this webservice with the FastAPI framework. The model serving is a separate web-service because it requires more RAM and CPU for

[3] https://docs.djangoproject.com/en/3.1/topics/security/

loading the AI models as well as storage for (temporarily) storing the videos and models than the web app which allows us to scale it independently of it.

- **Cloud storage**: For performance reasons, we store large binary data such as videos, images and AI models on Azure using Blob Storage. Currently, only videos uploaded from the platform are stored there. The videos on the NAS from ETH have not been migrated yet to Azure.
- **Model Retraining application**: For training new AI models. The models are trained with the darknet framework using the code written by the other UZH group. This web service is only up if admins want to train new models because the computational resources are very expensive compared to general virtual machines.

To provide a highly available and scalable platform, we manage the platform with Kubernetes, Docker, Terraform[4] and Kubespray[5]. The initial step is creating the infrastructure on which we are going to run our application. That means creating the virtual machines, disks, network connections and others. This is done using Terraform which supports multiple major cloud computing vendors, and therefore we are able to define the infrastructure we need using cloud vendor specific definitions in their respective *.tf* files. Previously we have used the Google Cloud Platform (GCP) for deployment of our infrastructure, and the infrastructure definitions for GCP can still be found in the *terraform* folder. However, we have eventually switched to Azure platform, and therefore the currently used Terraform definitions are for Azure platform. These can be as well found in the *terraform* folder.

The structure of the Azure infrastructure is rather complex compared to the GCP definitions. The three main sections are definitions of the network, virtual machines and resource groups. We define three virtual machines using the *linux_virtual_machine* resource types, resource groups using the *resource_group* resource types, and lastly the network using the *security_group*, *public_ip*, *network_inteface*, *route_table*, *virtual_network* and *subnet* resource types. More detailed information of what the variables mean, and which values are available can be found in the Terraform Azure cloud provider documentation[6].

The second step in creating our platform is deploying the Kubernetes cluster onto the previously created Azure infrastructure. This is done using the Kubespray project. The core idea of the Kubespray project is that developers have to define only very high-level variables, and everything else is taken care of. These variables are the type of infrastructure where we are deploying our cluster, in our case the Azure, and the network implementation we want to use. We are using the Flannel[7] implementation, however in our case this does not matter as we are not using any implementation-specific features. These two variables are defined in the *kubespray.sh* file.

After the Kubernetes is deployed onto the infrastructure we finally download the Docker images and Kubernetes configuration files, which is automatically done using the *initCluster.sh* file. Every microservice is packaged and self-contained using Docker and their respective Dockerfiles.

As can be seen in Figure 2, the Kubernetes cluster consists of three virtual machines where one of them is dedicated to running Kubernetes applications responsible for scheduling,

[4] https://www.terraform.io/
[5] https://kubespray.io/
[6] https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs
[7] https://github.com/coreos/flannel

managing and monitoring other applications and the network. The monitoring allows us to clearly see the usage of the computing resources and their status, e.g. running, crashed, stopped etc. This proved especially useful when debugging memory issues caused by the Model Serving application. The monitoring can be accessed on https://grafana.beelivingsensor.eu. However you need login details which are mentioned in the project *README.md* file.

Figure 2 also shows the data flow between those docker containers. Clearly, the central point is the webapp which accepts requests from the clients and returns templates ('pages') to them. To process these requests, the web-app relies on the database for retrieval and storage of data. It also interacts with the other two web-services, the model-serving app and the retraining app. For example, once the user has uploaded a video, the webapp requests the model serving app to analyze the video. Following a response, it will store the images in the cloud and the annotations in the database. In the case of the retraining app, the web app requests training of new models. This link has not been established yet in the platform but is envisaged by us. Unlike the images which are uploaded by the web-app to the cloud, the videos are directly uploaded by the clients which speeds up the upload and takes some of the burden off the web-app.

2.3 How to run the platform

Repository structure

We store all our code in in a GitLab repository[8]. The repository is structured as follows (documentation in comments):

```
├── darknet                    # Darknet source code
├── helmConfigs                # Helm configuration files
├── k8sConfigs                 # Kubernetes configuration files
├── modelServingApp            # Web service for model serving
│   ├── app                           # REST API code
│   ├── cfg                           # YOLO configuration files
│   ├── dev_requirements.txt          # Python dependencies for dev environment
│   ├── Dockerfile                    # Dockerfile for packaging Model Serving app
│   ├── requirements.txt              # Production dependencies
│   └── start_dev_server.sh           # Run model serving app in dev environment
├── retraining                 # Web service for retraining models
├── staticServer               # Web service for serving static files
├── terraform                  # Infrastructure definitions
├── webApp                     # Web service for webapp
│   ├── annotations                   # App for annotations (imagetagger)
│   ├── beewatch                      # Web application settings
│   ├── images                        # App for images (imagetagger)
│   ├── lib                           # Third-party libraries
```

[8] https://gitlab.com/beewatch/beewatch

```
│   ├── locale                        # Website language translations
│   ├── main                          # App for main
│   ├── nodejsSrc                     # Upload JavaScript functionality
│   ├── static                        # Static files used across apps
│   ├── templates                     # HTML templates used across apps
│   ├── containerRequirements.txt     # Python dependencies
│   ├── crontab                       # Scheduled jobs
│   ├── Dockerfile                    # Dockerfile for packaging web application
│   ├── manage.py                     # Django's CLI
│   ├── setup_cron_dev.sh             # Starting scheduled jobs in dev environment
│   ├── setup_postgis_dev.sh          # Setting up database in dev environment
│   ├── start_dev_server.sh           # Run web-app in dev environment
│   ├── startWeb.sh                   # Run web-app in production environment
│   └── storage.py                    # Azure cloud storage classes
├── buildImages.sh                # Build all Docker images and push to Docker Hub
├── gpuVMstart.sh                 # Deployment and initialization of model retraining
├── initCluster.sh                # Deployment of our microservices onto Kubernetes
├── kubespray.sh                  # Deployment of the Azure infrastructure
└── secrets.env                   # Environment variables for credentials, API keys, etc.
```

Generally, we created separate directories for each web service (*modelServingApp*, *retraining*, *staticServer*, *webApp*). Moreover, we have folders for Kubernetes and Terraform configuration files. The shell scripts in the root directory are used for deploying the Kubernetes in production.

For the webapp, we used Django's standard directory structure. Each app is located in a separate directory. Currently we have three apps: *annotations*, *images* and *main*. The *annotations* and *images* apps are from the ImageTagger[9] library. Views that we implemented are in the *main* app. In the future, we plan to merge those apps. For each app, views and templates are located in their respective folders. URLs are defined in *urls.py* and the database models in *models.py*.

Run instructions

Development web application

To run the platform locally, the *secrets.env* has to be created first in which the following environment variables have to be defined:

```
DJANGO_SECRET_KEY
DJANGO_PASSWORD
POSTGIS_PASSWORD
OPENWEATHERMAP_API_KEY
AZURE_STORAGE_CONNECTION_STRING
STORAGE_KEY
```

[9] https://github.com/bit-bots/imagetagger

To start the web-app locally, you can run the *Dockerfile* or execute the following command in the *webApp* directory:

```
$ sh start_dev_server
```

For the model serving app, we defined all endpoints in *app/main.py*. To run it locally, you can again run the *Dockerfile* or execute the following command in the *modelServingApp* directory:

```
$ sh start_dev_server
```

Note that you do not need to start the static server locally, because Django serves static files in the dev environment.

Production web application

As mentioned in the Architecture section, the process of deploying the production application starts by creating the infrastructure, deploying Kubernetes on that infrastructure, and then deploying our microservices.

For deployment, the user only needs to have the *secrets.env* file containing all the passwords and secrets, the private key used in logging into the virtual machines, and the *kubespray.sh* file. If all three files are present the user can execute the *kubespray.sh* file and initiate the process. The only thing the user has to do at the end, is check whether everything passed successfully. Unfortunately, we are not able to ensure deterministic deployment, and there is always a chance something will fail, and we are not able to recover from this error. There are countless discussions on the forums of projects which cause these problems, however there is no ultimate solution. If some part of the deployment process fails, we have designed the *kubespray.sh* script to be safely rerun, and it will retry the deployment again.

Retraining application

The only files needed for initiating the retraining of models are the previously mentioned \`secrets.env\`, private key used for logging into the virtual machine, and \`gpuVMstart.sh\` file.

The whole process starts with creating the virtual machine which has access to GPUs. Unfortunately, we were not able to automate this step yet, as every time we create a virtual machine with GPUs, we need to confirm Nvidia EULA for using their libraries, and we were not able to overcome this programmatically. Therefore, the users have to manually create the virtual machine using the Azure web interface, and when this is done, use the \`gpuVMstart.sh\` file to initialize the application inside the virtual machine. This will automatically compile the darknet binary using the source code in our repository, create configuration files for model training, and ask the web application which models should be trained. After each model is successfully trained, this can take anywhere between 12 and 72 hours, based on the configuration, the models are uploaded to the cloud storage, and registered as usable in the web application. After all of the models are trained, the virtual machine can be safely destroyed.

3 Conclusion

As part of our Master project, we have managed to create a fully operational platform that includes all elementary functions that were initially defined in the proposal. Nevertheless, the platform is far from finished. In fact, we still have around 62 open issues[10] in our product backlog and many unwritten feature requests like the aggregation of plant, agriculture and satellite/drone data and the detection of pollen and varroa. We have acquired many new skills in computer science, especially in the area of cloud computing, continuous integration/continuous delivery and REST APIs. Also on the process-side, we became a well-coordinated team following a purely agile approach.

[10] As of 8. December