Georgi Yanev

Desktop Application for Drone Image Metadata Processing

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master’s Thesis

7 April 2022

PREFACE

Writing this thesis was an interesting opportunity to reflect on the process of creating this particular piece of software dealing with drone image metadata processing. It has been quite the adventure, both the project itself, as well as describing it in this document. I can absolutely say I learnt a lot about image metadata and image processing while working on this project and obsessing over UI and making sure that the elements we designed interact in a nicely flowing, easy to use way, and allow the user to accomplish whatever they need to accomplish has been one of the top priorities.
Receiving a lot of feedback during the process has been instrumental to the completion of the modules and a lot of the improvements added were a result of direct feedback from end users, which was very appreciated. In many cases those improvements were not necessary exactly what the users communicated out loud, but rather what they expressed they needed, so some distilling and deep understanding of what we were really trying to solve was necessary.

Dedicated to my son, Anton, who was at times writing his “own thesis” in a notebook, alongside me.

Helsinki, 7 April 2022
Georgi Yanev

Abstract

Author:	Georgi Yanev
Title:	Desktop Application for Drone Image Metadata Processing
Number of Pages:	53 pages
Date:	7 April 2022
Degree:	Master of Engineering
Degree Programme:	Information Technology
Professional Major:	Networking and Services
Supervisors:	Ville Jääskeläinen, Principal Lecturer

This thesis describes the design, development, and improvements of modules in an application dealing with extracting and processing metadata from drone images.
The application was developed with web front end technology stack (JavaScript, React) and was deployed as an Electron desktop application for Windows, Linux and MacOS.
Many different features around image metadata, image processing and image management were identified as requirements in terms of functionality needed to accomplish the tasks of ultimately combining the extracted metadata, creating, and exporting entities called datasets, comprised of the original images, as well as generated files containing information extracted from the images.
Like many living software projects, many features were the result of post implementation user feedback, as well as many iterations and attempting to solve the challenges around processing and rendering thousands of images in a performance effective fashion.
The end result of how the modules ended up working covers all requirements and adds functionality on top that was not designed initially but was welcomed by users.

Keywords:
drone, image, metadata, EXIF, XMP, processing, desktop, macos, linux, electron, react, javascript, redux, leaflet, open source, Exifr, github, beta, software release, ci/cd, continuous integration, data processing

Contents

List of Abbreviations

AR	Augmented Reality
CSS	Cascading Style Sheets
EXIF	EXchangeable Image File Format
GPS	Global Positioning System
HTML	Hyper-Text Mark-up Language
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
Lidar	Light Detection and Ranging
OA	Operator App
OS	Operating System
PDF	Portable Document Format
PNG	Portable Graphics Format
UI	User Interface
UX	User Experience
XMP	Extensible Metadata Platform

1 Introduction

Data is everywhere, including in images. Drone imagery of various electrical grid facilities and conductors, for example, has been used to assess wear and tear and the state and quality of equipment. Besides the purely visual aspect of image data, said images contain a lot of valuable metadata too. This includes but is not limited to timestamps, Global Positioning System (GPS) coordinates, information about the drone and camera gimbal's positioning (roll, pitch, yaw), and a lot more. This data can be very useful and informative, as it can be used to calculate and infer other information, calculations, and statistics, which in turn can be used to be displayed on dashboards, maps, and various reports so that the information could be visualised in a clear and easy way to be understood and acted on by humans.

1.1 Background

In the first quarter of the 21st century we live in a world of data and information. The trend of data generation has been growing exponentially over the years, as processing and storage have become cheaper and more accessible to everyday people. People generate tons of data with their actions on a daily basis, whether they like it or not. From their behaviour online being tracked by advertising agencies to collecting telemetry and other data from personal devices such as smart phones, location, photos, sending messages online, video calls and so on, it is truly a world of data.

And while Augmented Reality (AR) is something to still catch on and in the works, the world is becoming more and more digital. Digital Twins are something that has existed for a while and yet more and more companies are looking to utilise that technology and create digital twins of their assets. Whether the asset in question is a cruise ship in the middle of the Pacific Ocean, or an electrical grid somewhere on the west coast of the USA, the principles are the same - "scan" with Light Detection and Ranging (Lidar) the assets into point clouds and use those to render out a 3D digital representation. Many times, digital twins can be

implemented as “living digital twins”, meaning the digital representation of the asset is not only identical visually but also has other metrics provided by sensors in real time (temperature, live video feed, etc.).
One of the ways to acquire all necessary information and work towards creating a digital twin of an asset, is by flying drones and using the camera and/or Lidar sensors, among others to collect the necessary data.

1.2 Drone Images Metadata

After briefly touching on drone image metadata, it is important to consider what it is and why is it so special? The benefits of drone image metadata could be illustrated with a simple example. As already mentioned, drone images in most cases and at the very least contain GPS coordinates metadata, which represents the specific geographical position of the craft at the time when the image was taken. When a drone is flying around an object, taking photos at various angles and distances from the object in interest, there is some sort of a flight trajectory. And indeed, by extracting the necessary metadata (GPS coordinates in this case), creating a list of it, and thereafter plotting the coordinates on a map, the craft’s flight trajectory can be successfully calculated and visualised. This is one example of how raw data from the image metadata can be used to calculate and provide additional information, that may not be immediately obvious, but could certainly be useful in the grand scheme of things when working to visualise and understand flight paths.

1.3 Objective

The objective of this thesis is to produce new modules in a desktop application capable of processing and extracting image metadata from hundreds or thousands of high-resolution drone images. The application should be able to handle that in an efficient way with a simple and easy to use user interface, providing warning and error messages about any potential issues and actions

required from the user, to ultimately simplify and speed up the image processing pipeline. In addition, the application should be able to be run on MacOS, Windows and Linux operating systems, and it should be able to be updated in an easy and user-friendly fashion.

2 Solution Design

Prior to the beginning of this project dataset creation from images could be handled mostly with random scripts. It has become apparent that a better solution would be to have a dedicated module to handle that task, right from within the existing application, called Operator App (OA). And although there are a lot of details and specifics that go into the implementation, in a nutshell, the module should be capable of handling thousands of images with ease and while delivering a decent user experience (UX), extract metadata and use it to generate datasets.

2.1 Current State Analysis

At the start of the project, there was a basic Electron application, developed with React and Redux and vanilla JavaScript (no typing with TypeScript). Only two modules existed - the Payload module and the S3 Uploader module. Therefore, the application was only capable of interfacing with the hardware payloads, as well as upload only datasets to S3 (with a lot of limitations).

Figure 1. Operator App login screen.

The flow would be that the user would need to login as seen on Figure 1 and then be presented the UI as seen on Figure 2.

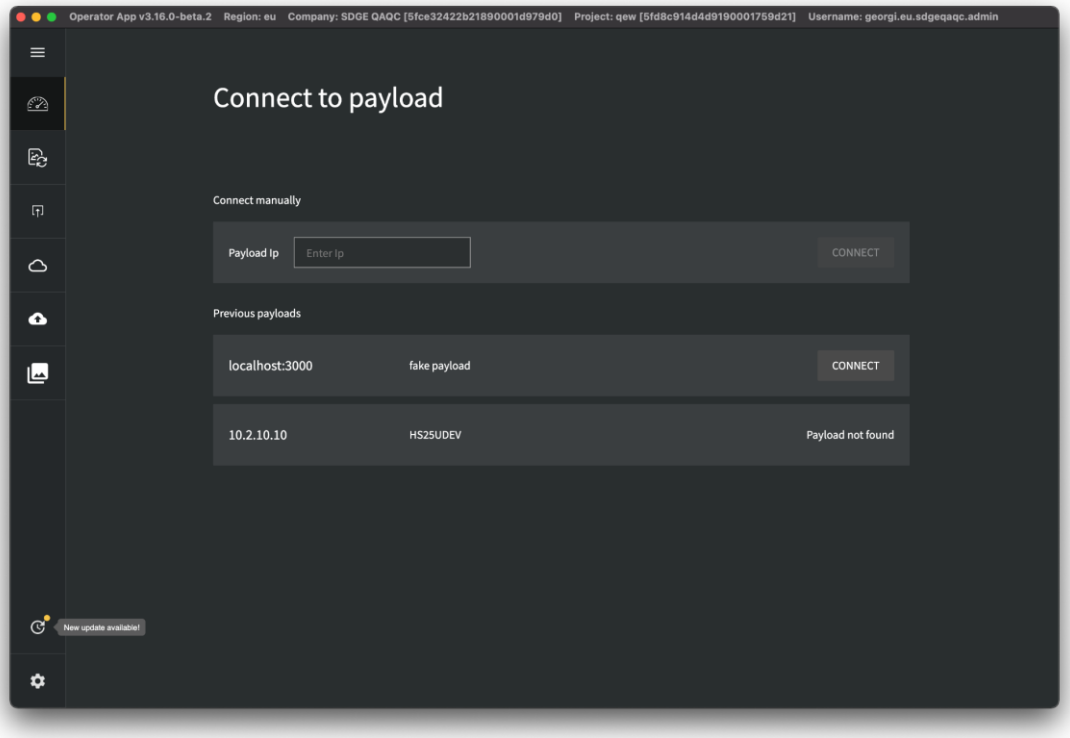

Figure 2. Main interface of the OA with the Payload module selected.

On the left-hand side was the collapsible menu with options to navigate to the different modules of the application. For lack of early screenshots, the example observed in Figure 2 contains all the new modules already added (the ones discussed and developed during the creation of this thesis).

2.2 Project Specifications and Necessary Improvements

The project specification for developing the drone image metadata extractor module was quite straightforward: provide an intuitive and robust user interface (UI) that allows the user to load, group and manage drone images and ultimately generate datasets to be used in the processing pipeline.

At the start of the project, quite a few necessary improvements had been identified:

- Set up a monorepo for the project and move code from different repositories
- Create a build pipeline capable of building the application and deploying the production artefacts to S3
- Allow seamless auto updates to be downloaded and installed
- Make the application completely cross platform: MacOS, Windows and Linux. Sign the application on MacOS to avoid issues.
- Migrate the codebase to TypeScript
- Add tests

2.3 Dataset Generator Module

When designing the solution for the Dataset Generator module several factors had to be taken into consideration. The output of the module had to be a dataset - an entity of many files - the images themselves, as well as a few metadata files created from reading the image metadata and consolidating the information. It had to support a lot of tools and options to allow the users to adjust how the datasets are created. The basic logic for creating a dataset is that the images of a datasets should be taken close to each other timewise. Assuming that each image taken within 2 minutes of the previous image belongs to the same dataset (or drone flight) already produces a great starting point for grouping files.

Figure 3. Example of images grouped together and time delta in seconds of each image versus the previous in the series.

Some of the initial Figma designs went a long way to describe how the UI should allow users to move images around between sets or delete them if they so desire, in order to adjust the datasets however they saw fit.

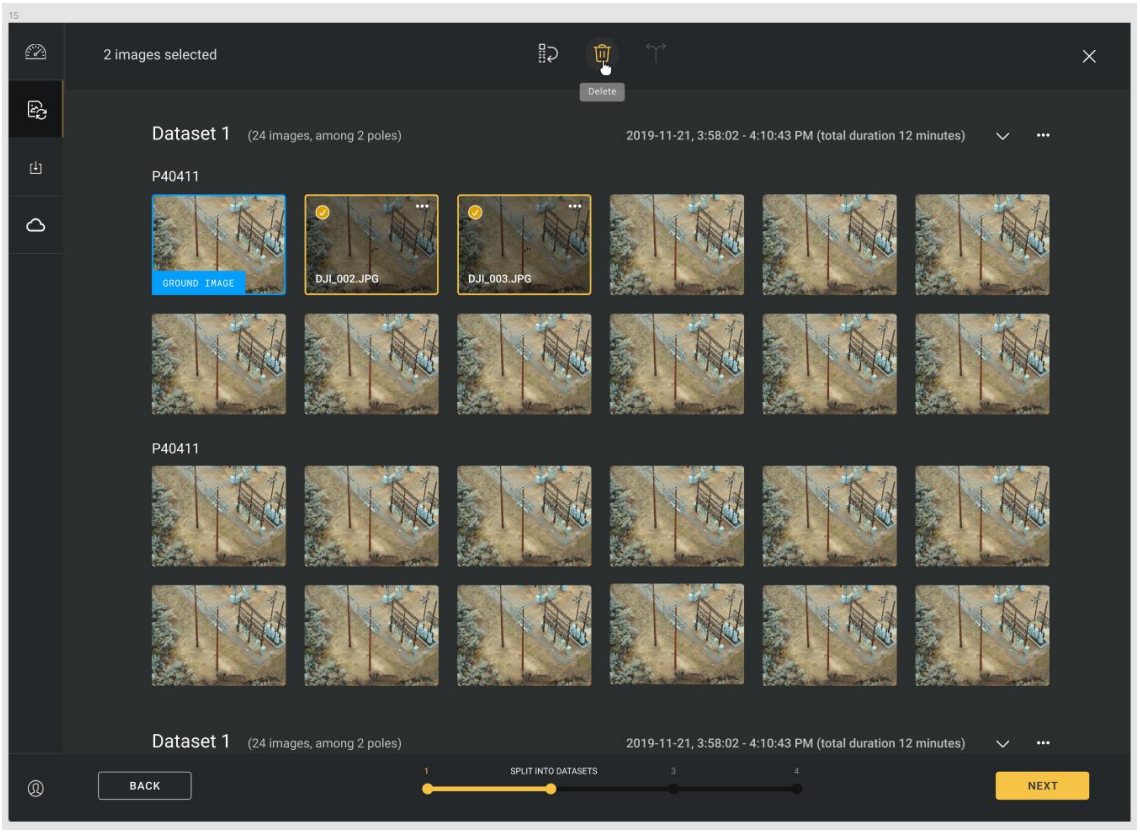

Figure 4. Figma design ideas for image management within the app.

Figures 4 and 5 communicate the desire of the initial design to have a robust UI for managing the images within the app, via different approaches – from multiselect work with multiple files at the same time, to having a quick access menu for each individual image. Some of the most important functionality, accessible from these menus is to delete images or move images between dataset groups.

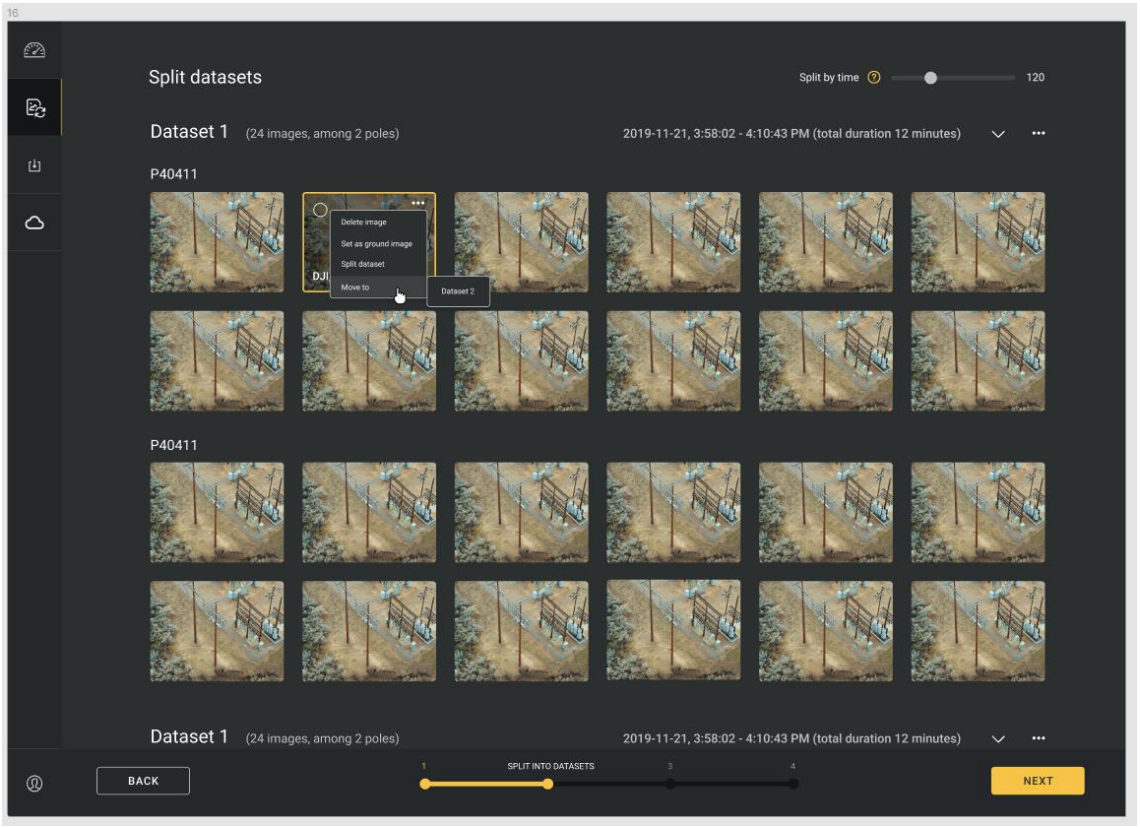

Figure 5. Figma design for the image menu.

Finally, rendering pins of the coordinates of the images, as well as pins for the assets on a map had to be tackled. This would allow users to assign images to assets. The initial design for this feature can be seen in Figure 6.

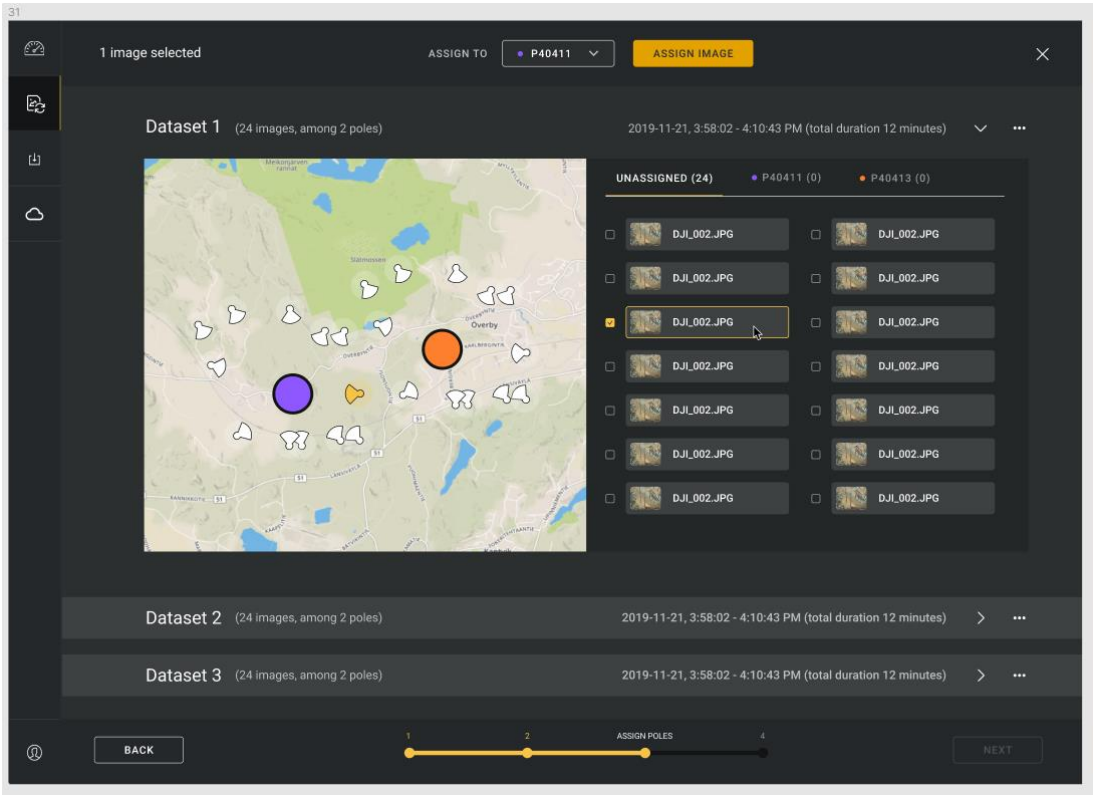

Figure 6. Figma design for map with image pins and assets rendered on it.

In addition to rendering the image and asset pins on the map, these components had to also support searching for assets if they were missing for whatever reason. The UI should have synced components where only the pins from the currently selected tab are rendered on the map, and thus when the user changes tabs on the right-hand side, only those images assigned to those assets should be visible on the map.

In the end, users should be able to see if there were any issues with the datasets they are creating, and if not, to press an export button and export the dataset with all the necessary images and metadata files written to a desired destination and ready for upload.

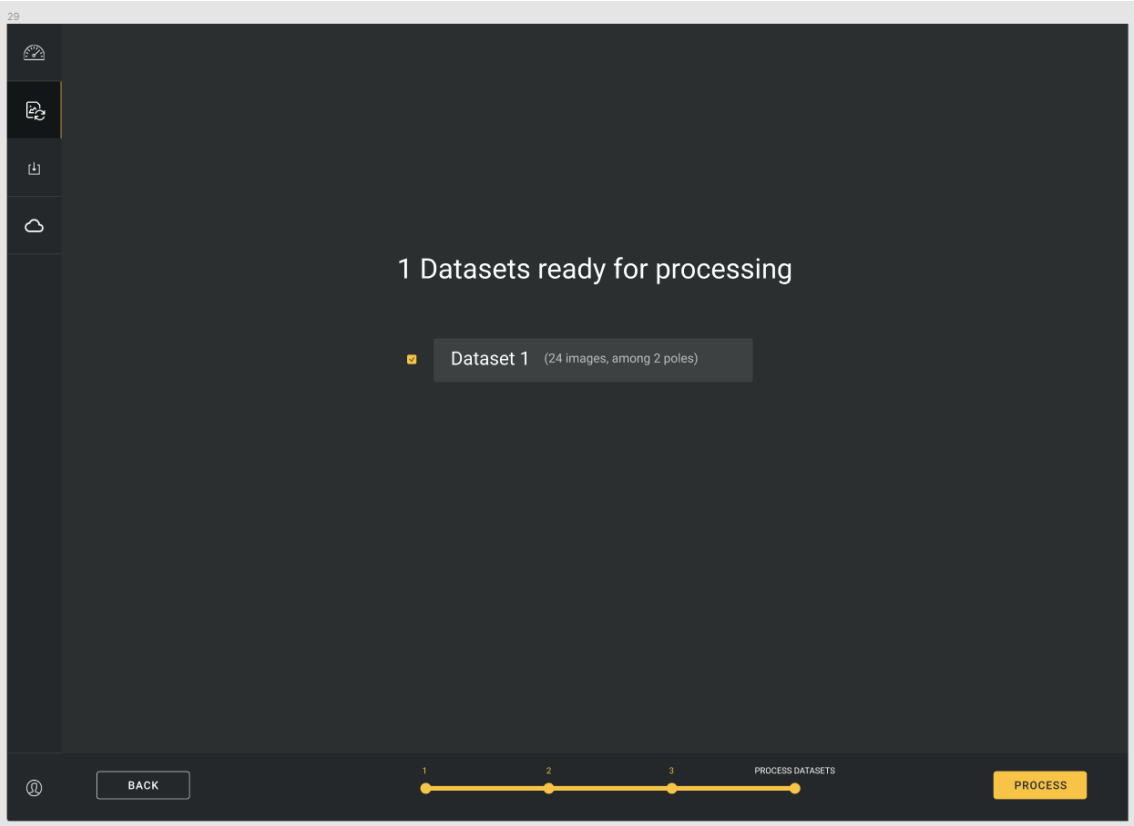

Figure 7. Figma design of the dataset ready for export view.

2.4 One Step Tool

The One Step Tool being a drastic simplification of the Dataset Generator module, would allow the end user to produce datasets even faster, with less clicks and of course is also therefor less robust and less configurable. In order to allow that to happen, an important assumption had to be made - the assumption that the user knows which asset they want to assign the images to. With that in mind, the rest of the process is very straightforward - anything the user drops in terms of images into the file drop input (or selects folders with the dialog) goes to be assigned to the selected asset.

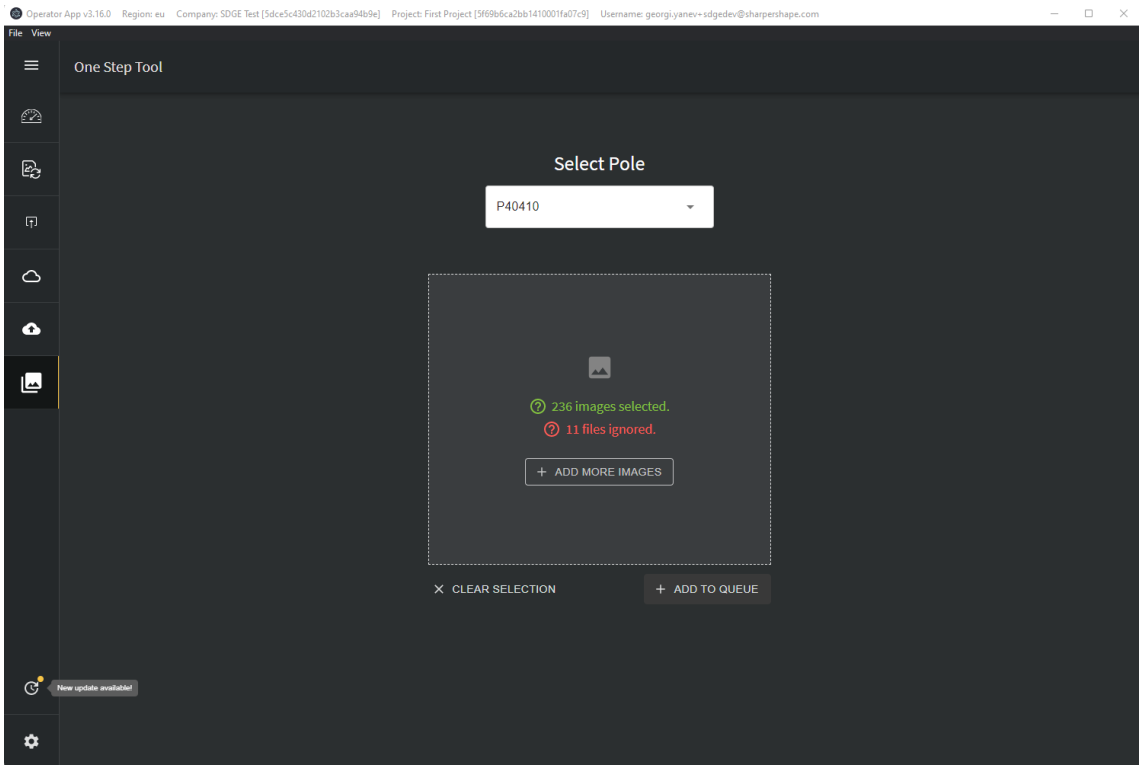

Figure 8. One Step Tool with asset selected and images dropped.

Therefore, the rest of the dataset creation process requires little to no additional interaction from the user - just drop the images to the asset and hit the export button, allowing for the processing of thousands of images instantly. These 2 modules go hand in hand, as design-wise behind the scenes, the processing and exporting functionality is the same and the application is refactored so that both modules use the same common functionality, only the flow and the UI being different.

3 Used Technologies

Most if not nearly all used technologies in this project are open source and available on GitHub. Some adjustments to the stack have been necessary over time and updating packages has been something that had happened periodically, in order to stay as current as possible, as well as avoid potential security issues.

3.1 Electron

Electron [1] is an open-source framework project that allows developers to use web / frontend technology stack (HTML, JavaScript, CSS) and build a desktop application with it, that can be deployed to Windows, MacOS, Linux. To do that, Electron combines the Node.js runtime and the Chromium open-source browser into a single package that enables users to implement, deploy and run the application on various machines and operating systems.

There are a lot of example open-source Electron applications on GitHub, and the technology has also been utilized by some closed source commercial projects, such as Slack, Visual Studio Code and Discord to name a few. Like most things in software development, there are PROs and CONs to using Electron for a desktop application, versus other solutions. Here are a couple of relevant comparisons just to illustrate the issues and challenges. For instance, in most cases, writing a native application for Windows or MacOS with their native coding tools and languages, would almost certainly be the best performing solution, as its closest to the bare metal and has less wrappers and abstractions around it. However, that would also require a team to know a lot more languages, tools, and specifics about each separate OS that they want to support. On top of that, the team would most likely need to maintain several different repositories and codebases, and each feature would have to be implemented into each separate codebase, using the specific language for that OS.

So, while Electron is perhaps not the best solution in terms of raw performance, it makes sense in a lot of other ways, in order to produce a still very decent

solution fast, maintain a single codebase and deploy to multiple platforms. Additionally, this provides the opportunity to consider sharing common functionality between this desktop application and other (web) applications and have a more consistent brand look and feel by utilising the same frontend UI libraries.

3.2 React

React [2] is a JavaScript library for building user interfaces. It has been around since 2013 and quickly gained a lot of popularity and for good reasons. Making dynamic components is straightforward with React and the framework is robust and capable with a lot of features. It is not the goal of this paper to drill into a complete comparison with other frontend frameworks, but it should be mentioned that because of the wide adoption of React, there are a lot of resources available to learn about different features and implementations, as well as a lot of other 3rd party open source supporting libraries.

3.3 TypeScript

TypeScript [3] is JavaScript with syntax for types. When TypeScript first came around it enjoyed a pretty much love or hate relationship with developers. There were those who immediately swore by static typing JavaScript and those who thought it was an unnecessary addition to the already heavy frontend tooling setup. As time went on, it seems that more and more people saw the benefits of TypeScript and more and more folks are coming around to it. In fact, in 2022 we just have a state 0 proposal of adding some version of typing in JavaScript itself in future versions.

While Electron and React were already in the Operator App project from the get-go, TypeScript was not, and at the beginning of this project I had no experience with it, but I worked on adding the TypeScript support and over a few months ported the entire codebase to be 99% TypeScript compliant. I learnt a lot from doing that and from using TypeScript over the past couple of years and really

learnt to appreciate static typing. It helps a lot with catching very early on potential errors, leads to better, higher quality code and tests and provides outstanding code completion when it comes down to calling functions and passing parameters to the calls.

3.4 Node.js

Node.js [4] is a JavaScript runtime built on Chrome's V8 JavaScript engine. Being utilized by Electron, Node.js allows developers to write some "back end" code, or essentially node.js code running in separate threads from the renderer UI process in Electron. Node.js itself provides access to the filesystem and other close to native APIs which help when dealing with reading and writing files for example.

3.5 Notable Open-Source Libraries

A few open-source libraries have been crucial for this project.

Sharp [5] is a high performance Node.js image processing library. It is one of the fastest options out there when it comes down to image processing, because it uses libvips behind the scenes. Sharp has been utilized for some image conversions from PNG to JPEG, for image blurriness detection checks, and for writing cardinal direction facing watermarks to images.

Exifr [6] is a library for extracting EXIF metadata (and also XMP and a few other formats too) from images. It does that very fast and efficiently, which is crucial when it comes down to extracting metadata from hundreds or thousands of images. More details on EXIF and XMP are available in the next chapter.

Leaflet [7] is an open-source JavaScript library for mobile-friendly interactive maps. It renders items quick and performs very well, while being a lightweight addition to the code base. Its supporting react-leaflet library makes it trivial to use in a React project.

Turf [8] is a library for advanced geospatial analysis. It allows for calculating areas from different geometries, creating bounding boxes, working with polygons and a lot more. One example of what Turf has been used for is to calculate the distance between an asset's location and the location of the drone at the time of taking the image. Another use case was to check if an image's location is within certain bounds (a specific bounding box perimeter around the asset).

3.6 EXIF and XMP Image Metadata

Most images taken by modern digital cameras in handheld devices or drones contain some form of metadata. There are some standards around what metadata there is and how it's shaped. EXIF (Exchangeable image file format) was created by Japan Electronic Industry Development Association (JEIDA) and is a standard widely used in most modern digital cameras [9].

There are a lot of specifics around how exactly the metadata is embedded into a JPEG image, which are out of the scope of this paper. As a brief explanation, by knowing the EXIF signature, key-value pairs of information from a specific segment of the image file can be extracted. This is what the above-mentioned open-source library Exifr does.

Fig. 3. EXIF Signature.

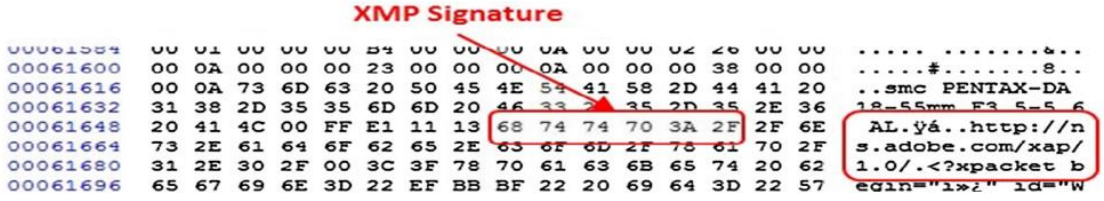

Fig. 4. XMP Signature.

Figure 9. Example of EXIF and XMP signatures in a JPEG file [10].

When it comes down to XMP (Extensible Metadata Platform), the idea is similar – standard metadata format, however, in this case developed by Adobe and based on XML. When it comes down to specifically extracting metadata from DJI drones, the XMP metadata is where craft specific information lies, such as roll, pitch, yaw values of the craft's orientation, for example.
Once the metadata is extracted, it makes available a wide list of EXIF metadata tags [11] and XMP metadata tags [12] which can be utilized for data processing.

Listing 1 and 2 show an example of how the extracted EXIF and XMP metadata looks like after extracting it from the image and displaying it in JSON (JavaScript Object Notation) form.


```
{
    "ImageDescription": "DCIM\\100MEDIA\\DJI_0021.JPG",
    "Make": "DJI",
    "Model": "FC6520",
    "Orientation": "Horizontal (normal)",
    "XResolution": 72,
    "YResolution": 72,
    "ResolutionUnit": "inches",
    "Software": "v01.09.2222",
    "ModifyDate": "2019-11-21T11:05:34.000Z",
    "YCbCrPositioning": 1,
    "XPComment": "Type=N, Mode=P, DE=Momo, BFM=AFS",
    "XPKeywords": "v01.09.2222;1.1.0;v1.0.0",
    "ExposureTime": 0.003125,
    "FNumber": 6.3,
    "ExposureProgram": "Normal program",
    "ISO": 100,
    "ExifVersion": "2.3",
    "DateTimeOriginal": "2019-11-21T11:05:34.000Z",
    "CreateDate": "2019-11-21T11:05:34.000Z",
    "ComponentsConfiguration": {
        "0": 0,
        "1": 3,
        "2": 2,
        "3": 1
    },
    "CompressedBitsPerPixel": 3.7377478015748995,
    "ShutterSpeedValue": 8.321,
```

```
"ApertureValue": 5.31,
"ExposureCompensation": 0.34375,
"MaxApertureValue": 1.69,
"SubjectDistance": 0,
"MeteringMode": "CenterWeightedAverage",
"LightSource": "Unknown",
"Flash": "No flash function",
"FocalLength": 45,
"FlashpixVersion": "0.1",
"ColorSpace": 1,
"ExifImageWidth": 5280,
"ExifImageHeight": 3956,
"ExposureIndex": null,
"FileSource": "Digital Camera",
"SceneType": "Directly photographed",
"CustomRendered": "Normal",
"ExposureMode": "Auto",
"WhiteBalance": "Auto",
"DigitalZoomRatio": null,
"FocalLengthIn35mmFormat": 90,
"SceneCaptureType": "Standard",
"GainControl": "None",
"Contrast": "Normal",
"Saturation": "Normal",
"Sharpness": "High",
"SubjectDistanceRange": "Unknown",
"SerialNumber": "fb5c7eeb903762d3bb03bcc0757c6a3e",
"LensMake": "Olympus",
"LensModel": "OLYMPUS M.45mm F1.8",
"GPSVersionID": "2.3.0.0",
"GPSLatitudeRef": "N",
"GPSLatitude": [
    32,
    47,
    54.3794
],
"GPSLongitudeRef": "W",
"GPSLongitude": [
    116,
    29,
    52.8558
],
"GPSAltitudeRef": {
    "0": 0
},
```

```
    "GPSAltitude": 1145.995,
    "latitude": 32.79843872222222,
    "longitude": -116.4980155
}
```


Listing 1. Example of extracted image EXIF metadata in JSON format.


```
{
    "about": "DJI Meta Data",
    "ModifyDate": "2019-11-21",
    "CreateDate": "2019-11-21",
    "Make": "DJI",
    "Model": "FC6520",
    "format": "image/jpg",
    "AbsoluteAltitude": 1146,
    "RelativeAltitude": 13.3,
    "GimbalRollDegree": 0,
    "GimbalYawDegree": -67.2,
    "GimbalPitchDegree": 7.1,
    "FlightRollDegree": -0.8,
    "FlightYawDegree": -71.4,
    "FlightPitchDegree": -0.2,
    "CamReverse": 0,
    "GimbalReverse": 0,
    "SelfData": "Undefined",
    "CalibratedFocalLength": 13636.364258,
    "CalibratedOpticalCenterX": 2640,
    "CalibratedOpticalCenterY": 1978,
    "RtkFlag": 0,
    "Version": 7,
    "HasSettings": false,
    "HasCrop": false,
    "AlreadyApplied": false
}
```


Listing 2. Example of extracted image XMP metadata in JSON format.

3.7 Monorepo setup with Lerna

The monorepo pattern is a popular one for managing more complex projects with Git. It basically allows to “break down” the project into several chunks, usually called packages and use and link those as dependencies in the parts of the application that require them. The open-source package Lerna [13] helps with managing the workflow of maintaining a multi-package repository and makes it easy to get started and develop quickly without any extra hassle.

Figure 10 below shows and example of the Lerna setup. All sub packages of the project are located within the packages directory and each package essentially acts as its own entity and has its own package.json file, can be developed, built,

imported, and used wherever necessary. The main package.json in the root of the project contains lerna scripts that are capable of booting up in parallel specific packages during development.

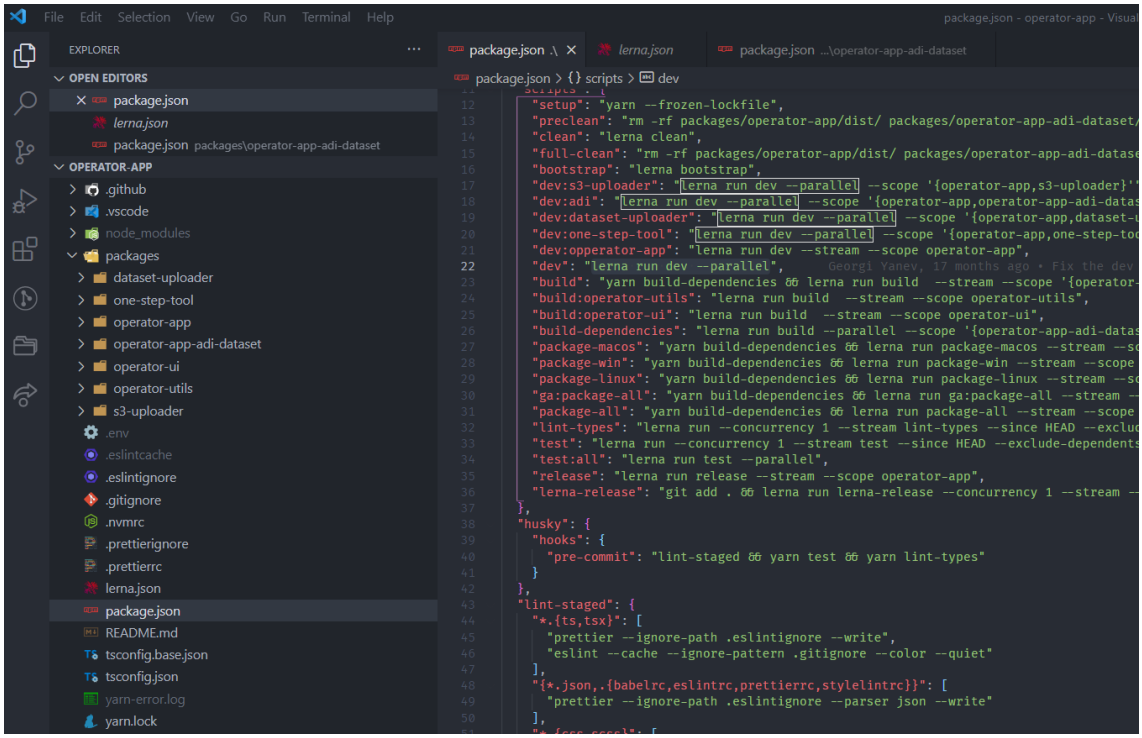

Figure 10. Lerna setup. Packages directory for all sub modules and lerna scripts in package.json to run across specific packages.

This setup is robust and allows for example to share code from the operator-utils package (which contains common utility functions) across the rest of the packages.

3.8 Testing with Jest

Testing a software project is important to make sure any potential bugs and errors are caught and never make it to the production builds. Jest is a popular and reliable JavaScript testing framework.

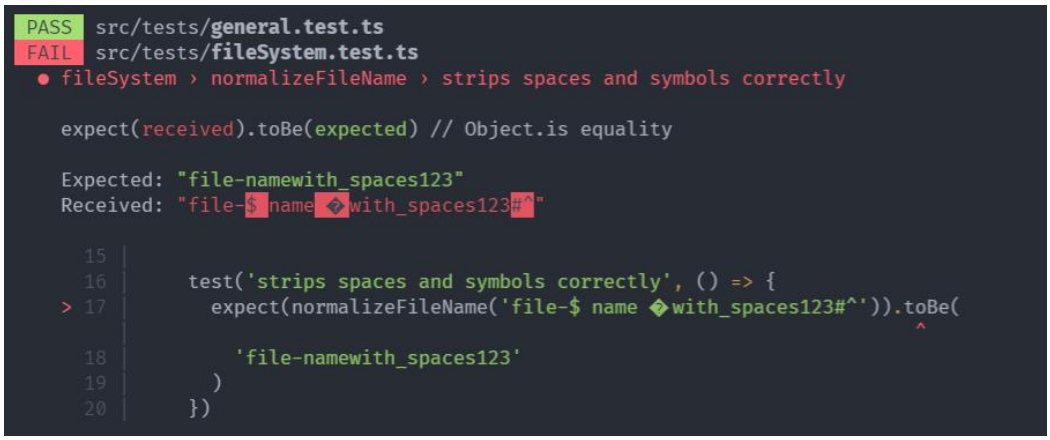

Figure 11. Jest tests failing.

Figure 11 above shows an example of unit tests failing for a specific utility function. Trying to provide a solid test coverage is something that had been improved a lot during this project. Figure 12 below provides an example when all tests pass successfully. It is possible to run tests in “watch” mode such that as the application is being developed, tests run continuously, and the status report is immediately available to the developer.

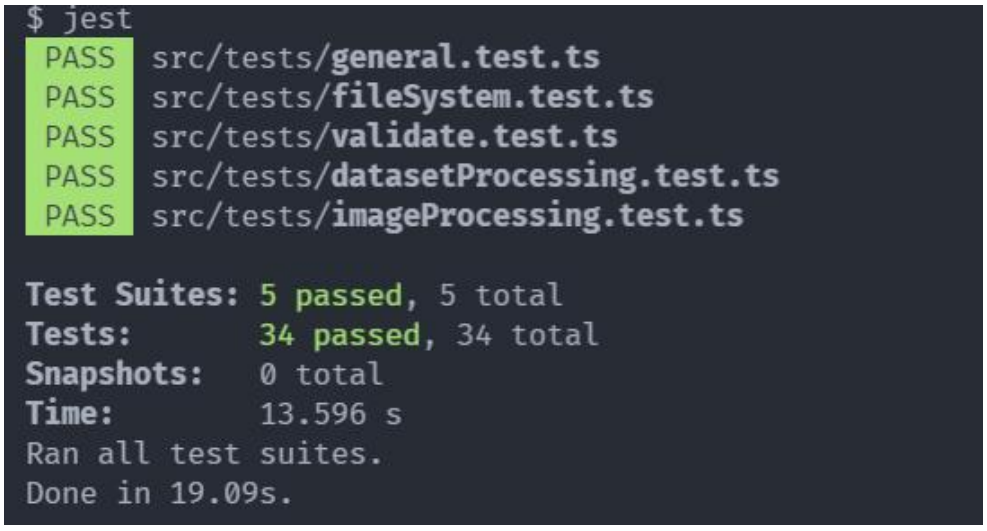

Figure 12. Jest tests passing.

Especially early on the project had been lacking on tests and as this implementation progressed, more unit tests were added to ensure higher code quality.

4 Continuous Deployment

Improving the continuous deployment of the OA software was identified as one of the necessary improvements in the current state analysis section. In the beginning of the project all builds and deploys had to be done manually. Additionally, because of specifics around how the project is compiled with native dependencies, it meant that the MacOS build had to be built on a Mac, Windows on Windows, and Linux on Linux. As work started on this feature, GitHub Actions was identified as a simple and robust solution to cover the requirements.

4.1 GitHub Actions

There are different continuous integration solutions, and GitHub Actions was one of the newcomers to the space. It was simple to configure and seemed to be able to handle what needed to be done for the Operator App project. Figure 13 shows the entire configuration.

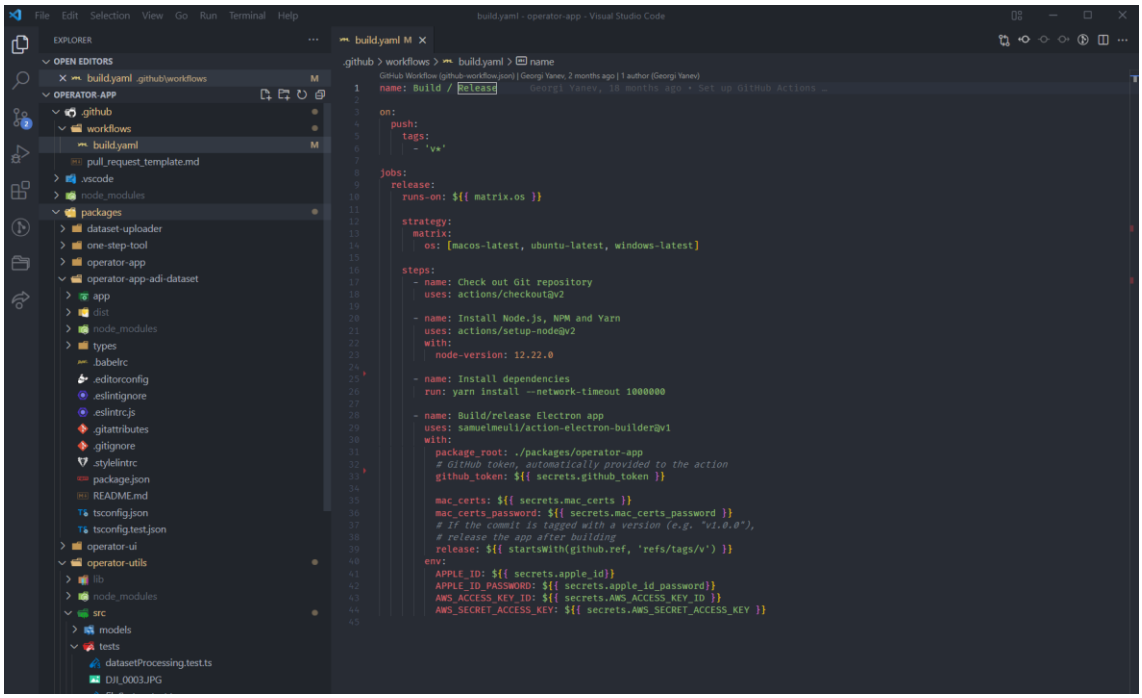

Figure 13. GitHub Actions configuration.

GitHub Actions config files live in the `.github/workflows` directory of a project and each yaml file can describe one specific GitHub Action. In the example above, the only action is the build action.

Actions are then accessible from the Actions tab in the project's repository, where more information about runtimes and status is available, as well as logs of the entire pipeline running.

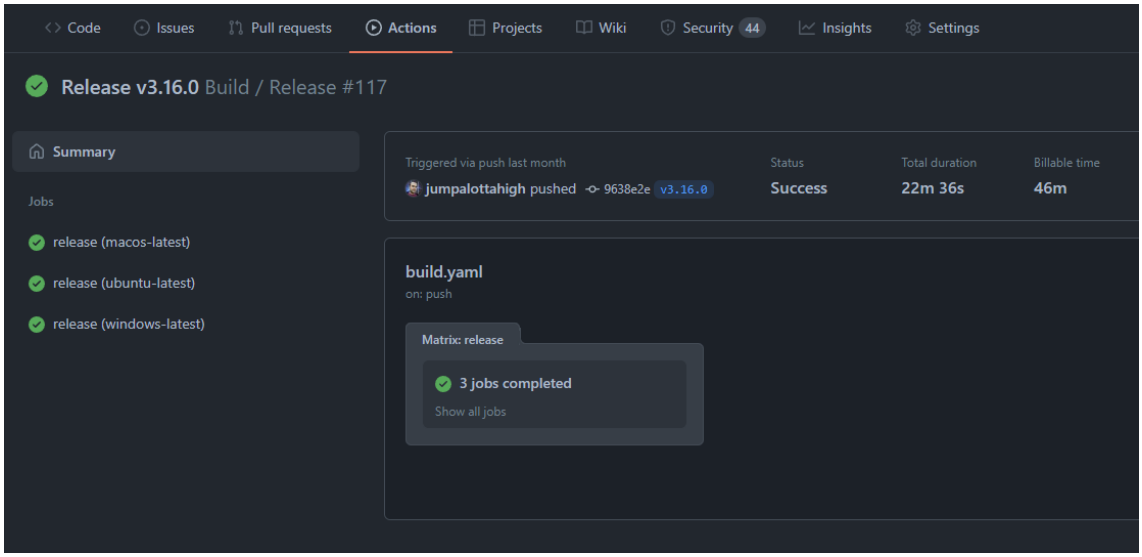

Figure 14. GitHub Actions UI.

4.2 Multi-Platform Builds

Getting started with multi-platform builds and running said builds in parallel, meaning the app gets built for 3 separate operating systems each built at the same time the build process starts, is as simple as defining the operating systems necessary as a matrix in the strategy build key in the config file for the action.

```
strategy:
  matrix:
    os: [macos-latest, ubuntu-latest, windows-latest]
```

This means that 3 docker containers will boot up and each will build the app for the specific OS. For this to work, several secrets need to be set up in the repository's settings. Specifically, for MacOS different certificates and credentials had to be provided to produce valid application builds.

```yaml
    strategy:
      matrix:
        os: [macos-latest, ubuntu-latest, windows-latest]

    steps:
      - name: Check out Git repository
        uses: actions/checkout@v2

      - name: Install Node.js, NPM and Yarn
        uses: actions/setup-node@v2
        with:
          node-version: 12.22.0

      - name: Install dependencies
        run: yarn install --network-timeout 1000000

      - name: Build/release Electron app
        uses: samuelmeuli/action-electron-builder@v1
        with:
          package_root: ./packages/operator-app
          # GitHub token, automatically provided to the action
          github_token: ${{ secrets.github_token }}

          mac_certs: ${{ secrets.mac_certs }}
          mac_certs_password: ${{ secrets.mac_certs_password }}
          # If the commit is tagged with a version (e.g. "v1.0.0"),
          # release the app after building
          release: ${{ startsWith(github.ref, 'refs/tags/v') }}
        env:
          APPLE_ID: ${{ secrets.apple_id}}
          APPLE_ID_PASSWORD: ${{ secrets.apple_id_password}}
          AWS_ACCESS_KEY_ID: ${{ secrets.AWS_ACCESS_KEY_ID }}
          AWS_SECRET_ACCESS_KEY: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
```

Figure 15. GitHub actions build workflow yaml file.

4.3 Auto Updates Implementation

Auto updates are important for a great user experience. Pushing updates to the user and asking to install and restart the app is a common way of handling updates and new app releases. This starts with the release process, and as can be seen in Figure 15 above, anytime a version tag is pushed, the release process kicks off. This builds the application artifacts for the different operating systems and uploads the new versions to a bucket in AWS S3.

When the app starts up it checks and notifies the user of a newer version is available. It is possible to opt in and auto download the updates in the background and then only prompt the user to restart the application to update. At this moment,

this is not enabled and when the user clicks the update button, only then the download is carried out.

It should be mentioned that the application also supports a separate release channel for Beta versions. This is an opt-in feature and can be enabled from the Settings dialog.

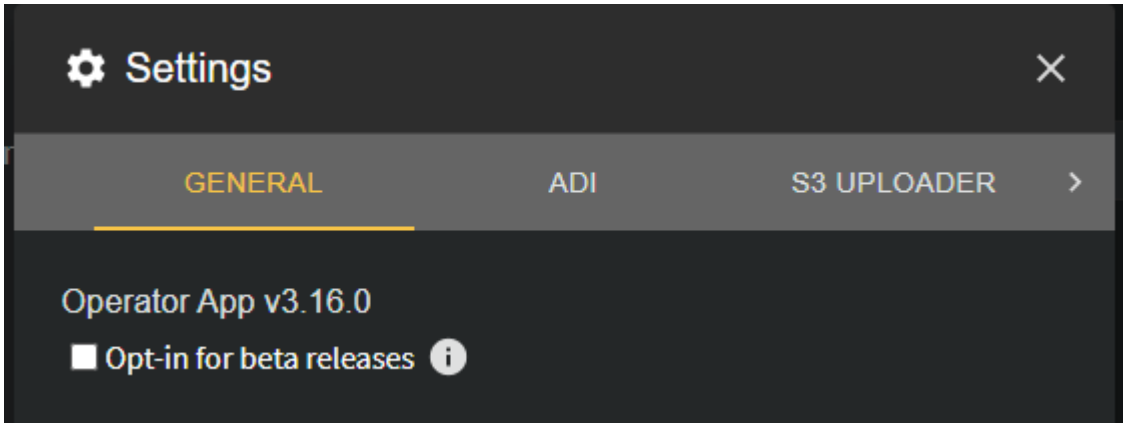

Figure 16. Settings dialog and beta opt-in option.

Once done, the user can also see beta releases in the update notifier. Additionally, rollback from beta to latest stable release is supported and can be done by simply disabling the beta opt-in and then installing the “update” that pops up – the latest stable release.

5 Developing Dataset Generator Module

The Dataset Generator Module was developed with a step-by-step wizard-like approach as discussed in the solution design section.

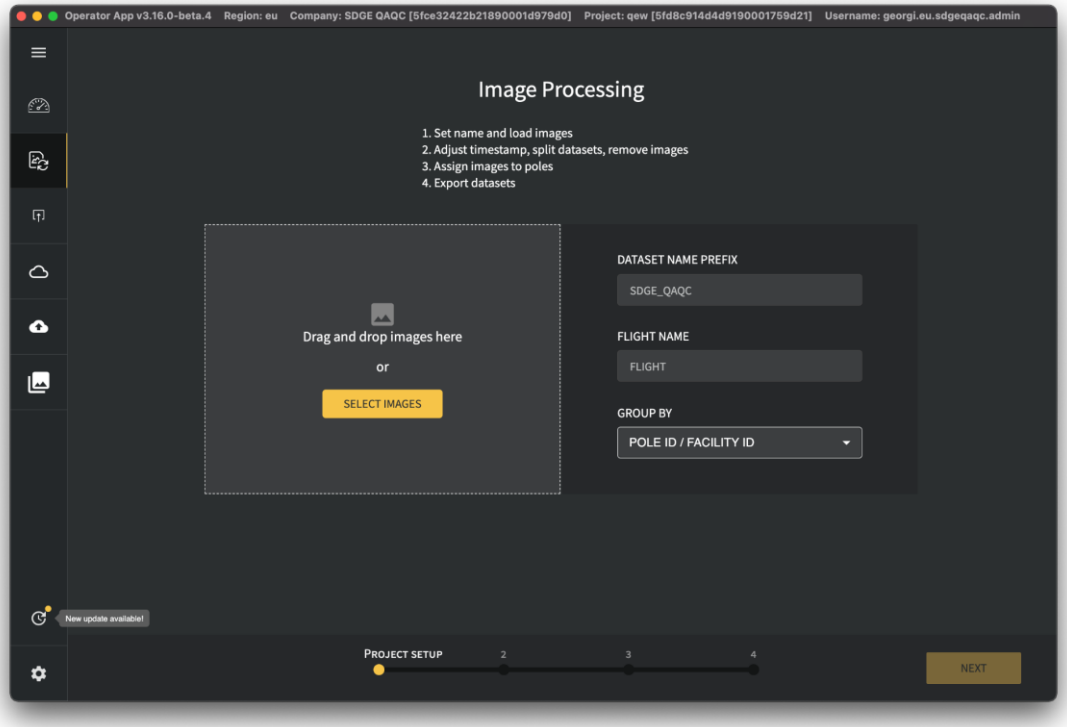

Figure 17. Dataset Generator Module starting screen.

On the first step of the wizard the user can set the dataset name, flight name and grouping type and drop images or select a folder to be loaded up. Any selection is traversed recursively down until a complete file list is returned. The application checks for specific file types, making sure only compatible image types are accepted and processed. During the development process because of specific requirements, support for accepting PNG images, and converting them to JPEG, was also implemented.

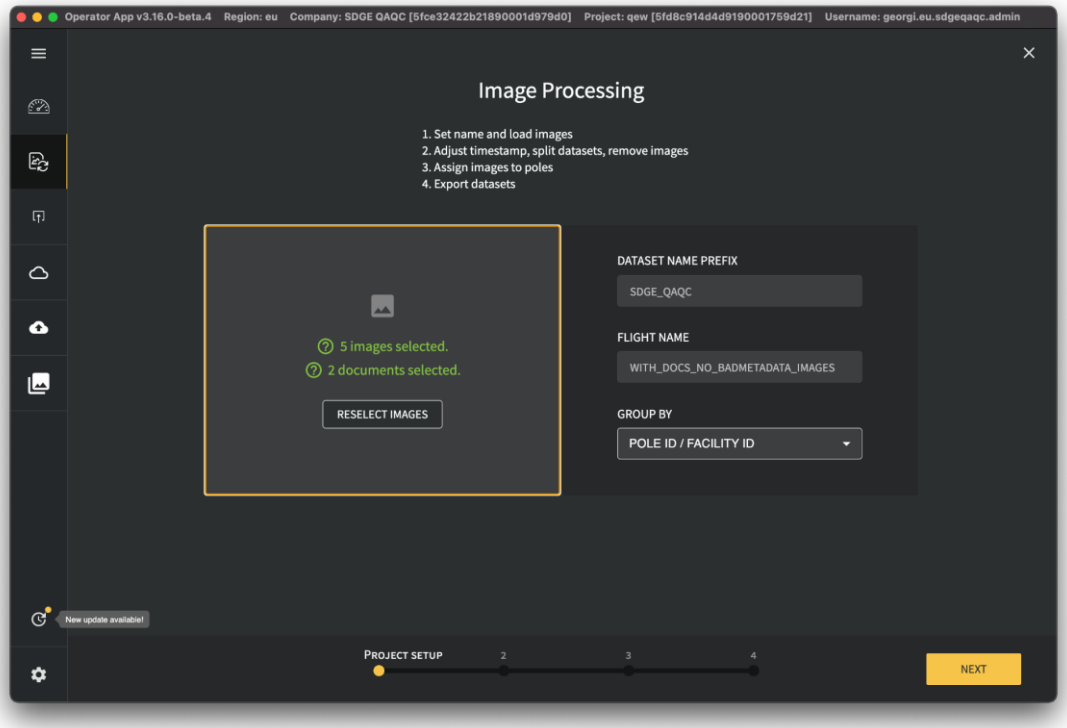

Figure 18. PDFs are also an accepted file type.

Finally, the OA also accepts PDF files to be selected and exported with the dataset.

When the user presses the next button on this first step, this is where the majority of the magic and computation happens, so the application display a loading spinner while doing the work and processing the images.

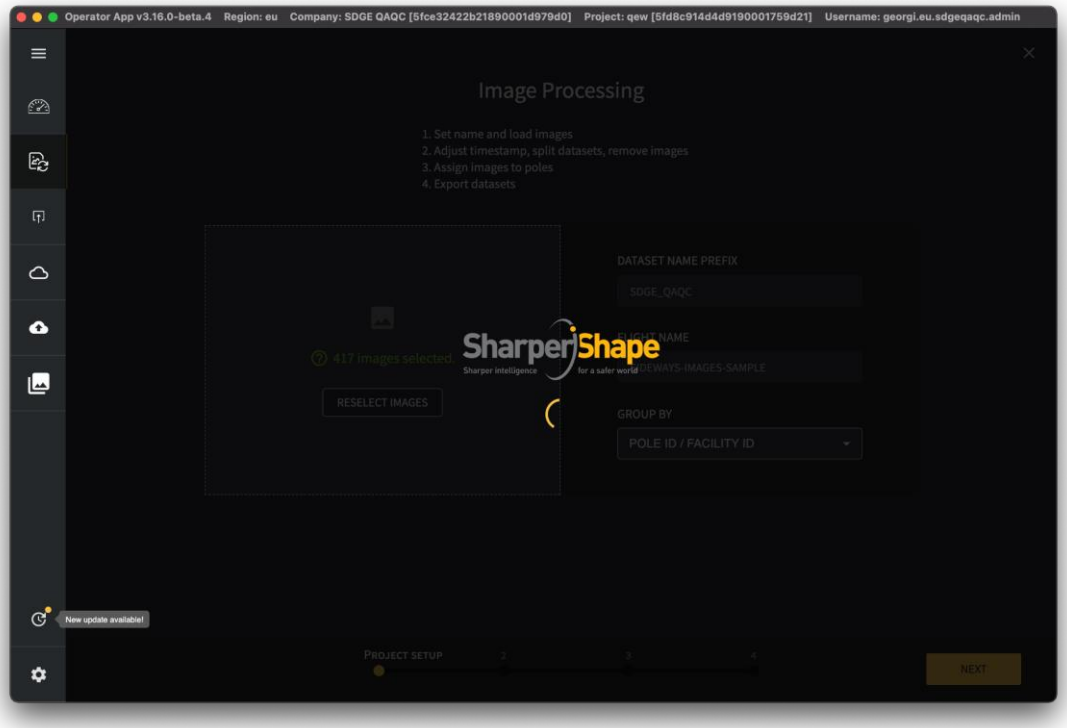

Figure 19. Loading screen while extracting image metadata and organizing images into datasets in memory.

5.1 Extracting Image Metadata

Extracting image metadata was briefly touched in the EXIF section above. By utilizing the EXIFR library it is possible to extract the EXIF and XMP metadata in one go as all the images are iterated.

At this point is when it is first possible to note if an image is corrupted or has any type of bad metadata (be it GPS, date, or any other important field). Bad metadata images are flagged as such and that information is passed along for dealing with later. It is unlikely but nevertheless still possible to have two images with the exact same timestamp, and so that case is handled as well to prevent overwriting existing valid entries. GPS information is extracted, degrees are converted to decimals and then validated. If validation fails, the image is flagged as a bad metadata one. See Listing 3 below.

```
// get GPS data from exif
let latitude = null,
  longitude = null

const gpsLatitude = getExifField(imageExifData, 'GPSLatitude') as
  | string
  | number[]
const gpsLatitudeRef = getExifField(
  imageExifData,
  'GPSLatitudeRef'
) as string
if (
  gpsLatitude !==
    DATASET_GENERATOR_CONFIG.DEFAULT_VALUE_FOR_MISSING_EXIF_FIELD &&
  gpsLatitudeRef !==
    DATASET_GENERATOR_CONFIG.DEFAULT_VALUE_FOR_MISSING_EXIF_FIELD &&
  typeof gpsLatitude !== 'string'
) {
  latitude = convertDegreesToDecimals(
    gpsLatitude[0],
    gpsLatitude[1],
    gpsLatitude[2],
    gpsLatitudeRef
  )
}

const gpsLongitude = getExifField(imageExifData, 'GPSLongitude') as
  | string
  | number[]
const gpsLongitudeRef = getExifField(
  imageExifData,
  'GPSLongitudeRef'
) as string
if (
  gpsLongitude !==
    DATASET_GENERATOR_CONFIG.DEFAULT_VALUE_FOR_MISSING_EXIF_FIELD &&
  gpsLongitudeRef !==
    DATASET_GENERATOR_CONFIG.DEFAULT_VALUE_FOR_MISSING_EXIF_FIELD &&
  typeof gpsLongitude !== 'string'
) {
  longitude = convertDegreesToDecimals(
    gpsLongitude[0],
    gpsLongitude[1],
    gpsLongitude[2],
    gpsLongitudeRef
  )
}

// validate lat and lng data
if (!isValidLongitude(longitude) || !isValidLatitude(latitude)) {
  badMetadata = true
}
```

Listing 3. Extracting, converting, and validating GPS metadata from the images.

Finally, an object with all the extracted metadata is returned for each image, as shown in Listing 4 below.

```
const parsedImageData = {
  [dateTime.getTime()]: {
    id: uuid.v4(),
    dateTime,
    parentDirectory,
    lensFocalLength,
    flightPitchDegree,
    flightRollDegree,
    flightYawDegree,
    fullPath: image,
    gimbalPitchDegree,
    gimbalRollDegree,
    gimbalYawDegree,
    latitude,
    lensType,
    longitude,
    make,
    model,
    relativeAltitude,
    serialNumber,
    badMetadata,
    imageBase64Thumbnail,
    absoluteAltitude,
    sensorResolutionH,
    sensorResolutionV,
    hasCardinalDirectionWatermark,
    exifOrientation,
    thumbnailRotationDegrees,
  },
}

return parsedImageData
```

Listing 4. Extracting, converting, and validating GPS metadata from the images.

5.2 Combining Extracted Metadata into Datasets

The first step is to sort all the images in the array by their timestamp value in chronological order. The dataset creation logic is based on the amount of time passed since the previous image was taken. This is a value adjustable by the user, but the default value which works for most cases is 120 seconds. Using this assumption that images taken within 120 seconds of each other fall within the same flight / dataset, it is possible to break the entire array of all images into separate datasets. During this iteration all bad metadata images are also filtered out into a specific dataset. This step ends with a validation of each dataset where issues can be flagged.

5.3 Additional Image Management Options

Once all the image processing work is done, the user is presented the view as seen in Figure 20.

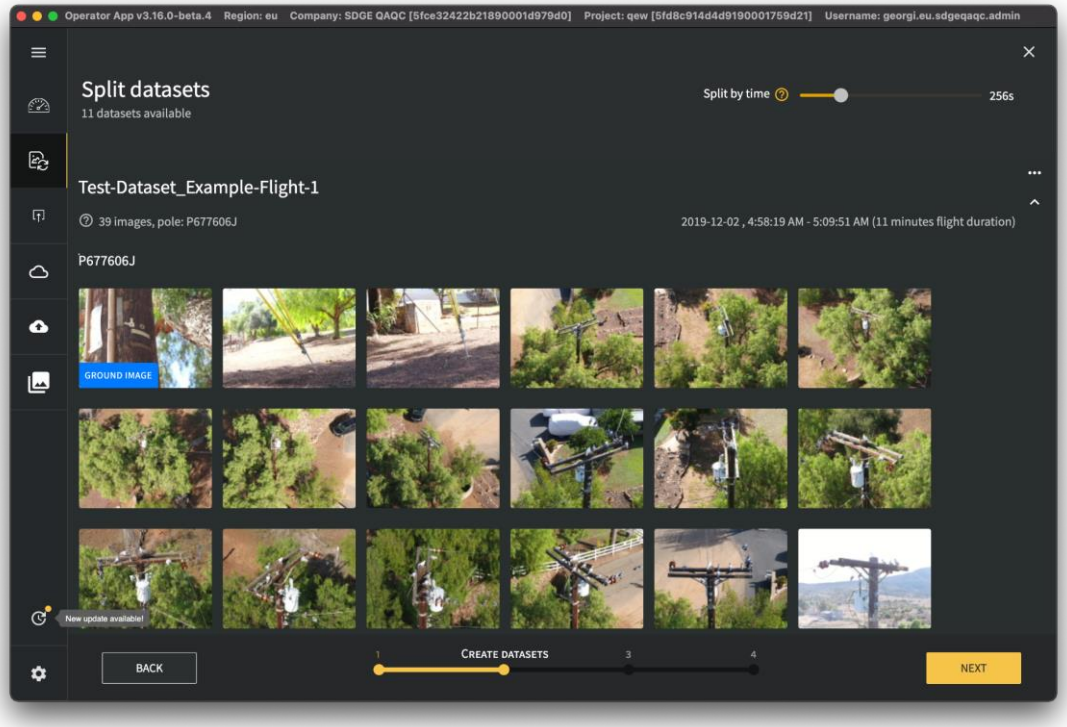

Figure 20. Dataset creation view.

The easiest way to re-adjust how datasets are created from the available images, is by using the slider at the top right. That will change what amount of time (in seconds) between images being taken should be considered for the images to still belong to the same dataset.

Once the time delta is adjusted and the user is satisfied with the preliminary grouping into baseline datasets, it is possible to do even more fine-tuning and further adjustments.

Figure 21. Dataset information tooltip.

Each dataset is a collapsible element and displays some of the most important pieces of information at a glance - the number of images in the dataset, the poles / assets that the application suggests the images belong to, date, flight start and flight end timestamps, as well as flight duration. If flight duration values do not look nominal, they could be used as an easy indicator that something may be wrong with the dataset or the image metadata.

To further adjust and manage datasets and images the user can: remove a dataset, merge 2 datasets together, as well as run a check for blurry images or add cardinal direction watermarks to images. These options are available by clicking on the 3 dots to expand the options menu.

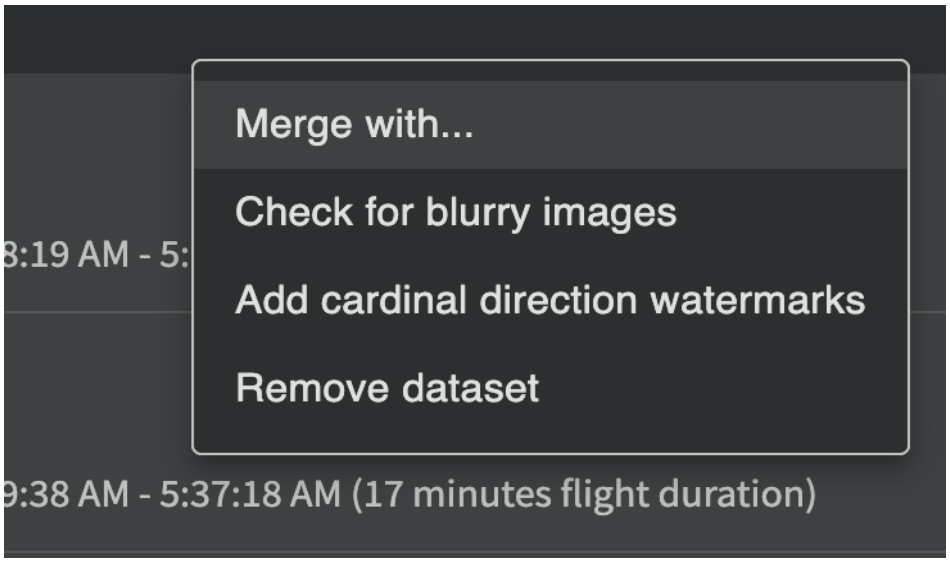

Figure 22. Dataset options menu.

Merging datasets together when done via the corresponding option opens up a submenu that lists all other datasets, so the user can choose where to merge this current dataset.

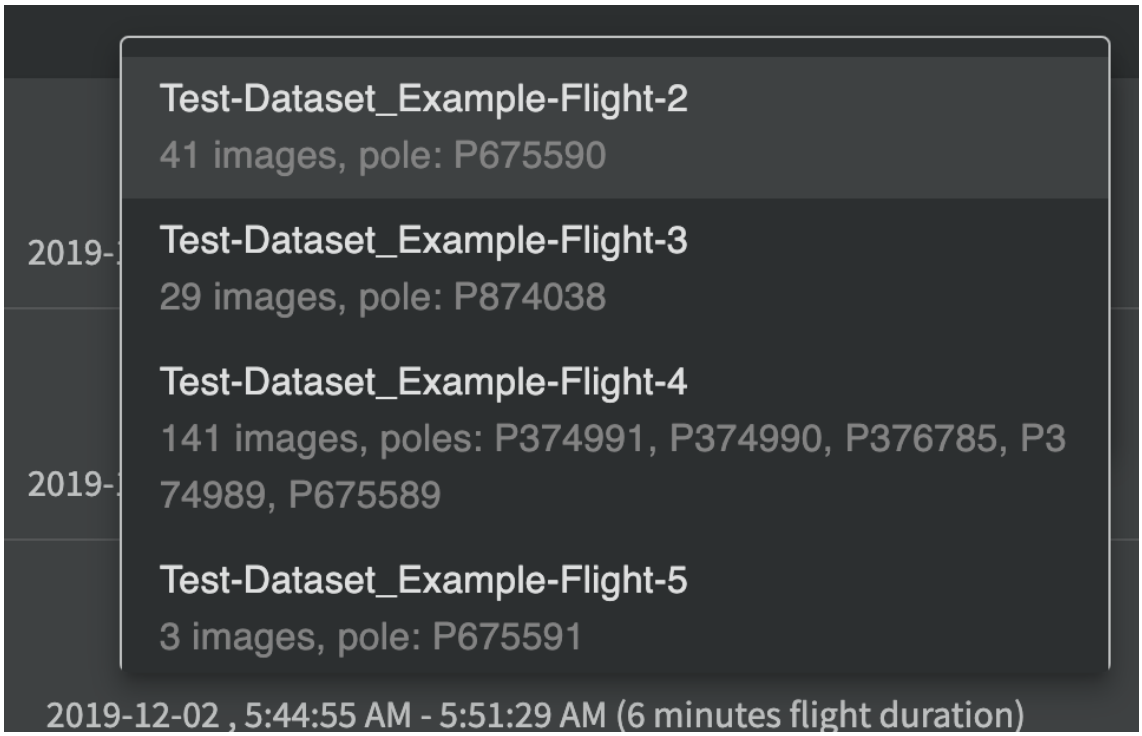

Figure 23. Merge dataset destinations.

The blurry image detection was an interesting feature to work on and I had to collaborate and reach out to the author of the sharp image processing library and am happy that because of my request that functionality is now available baseline in sharp.

The cardinal direction watermark feature places a watermark in the top right corner of the images with the letter of the cardinal direction they are facing.

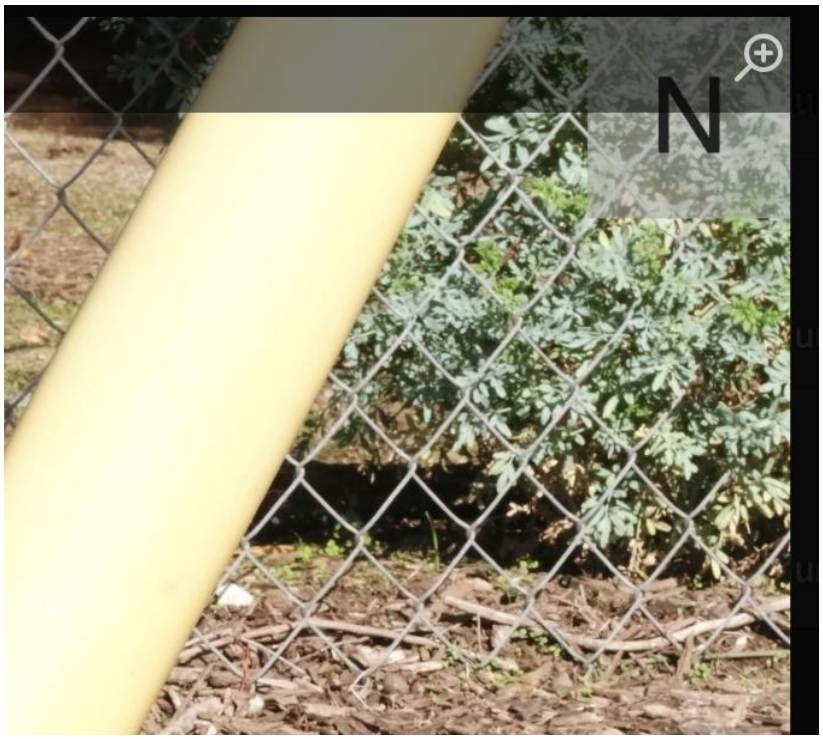

Figure 24. Image with a cardinal direction watermark for North facing.

Hovering over an image displays its file name, an empty circle icon for selecting the image and a 3-dot icon for the image menu.

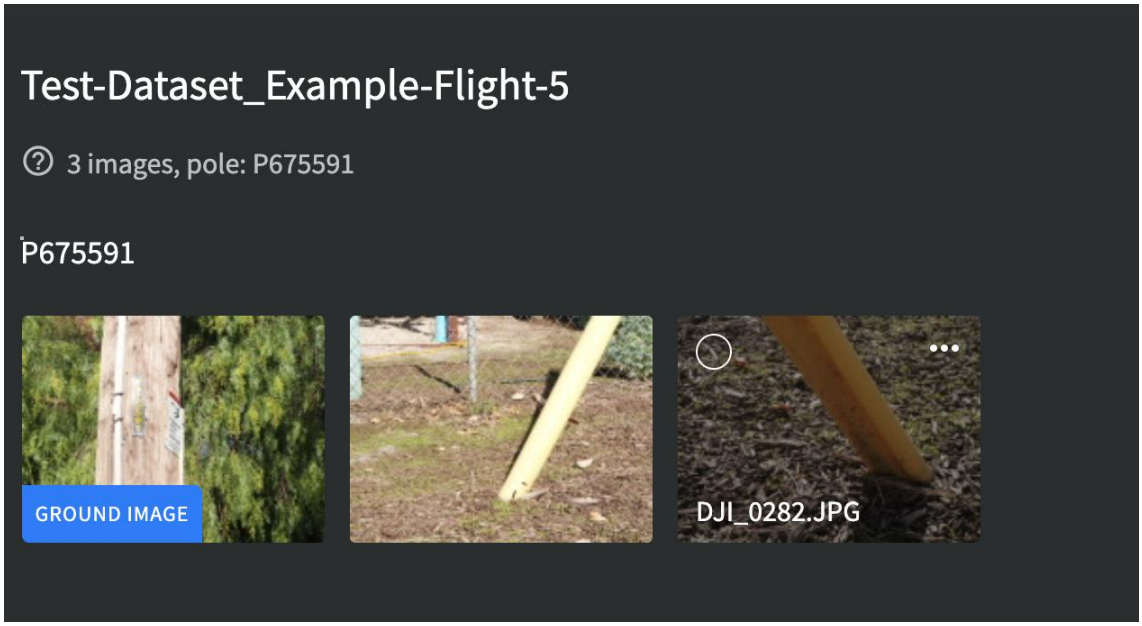

Figure 25. Image on hover UI.

The image menu contains the following options:

- Set an image as a ground image, in case the pre-assigned default is not a good fit
- Add a cardinal direction watermark for only this image
- View the EXIF and XMP metadata of the image
- Rename or remove the image
- Move an image to another dataset

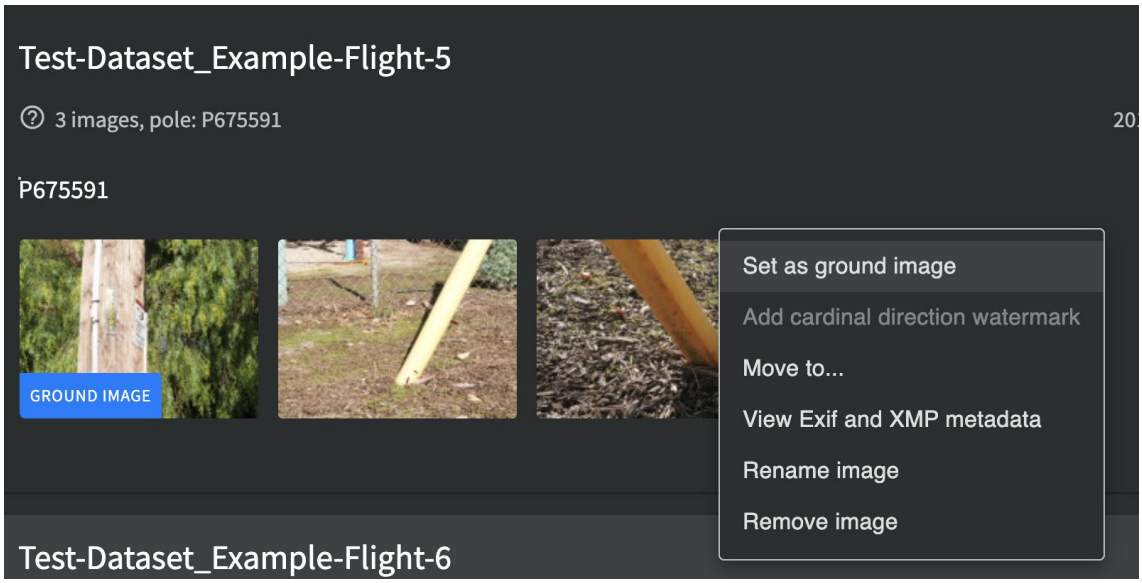

Figure 26. Image options menu.

Clicking on “Move to…” displays a submenu with a list of all valid datasets the image could be sent to.

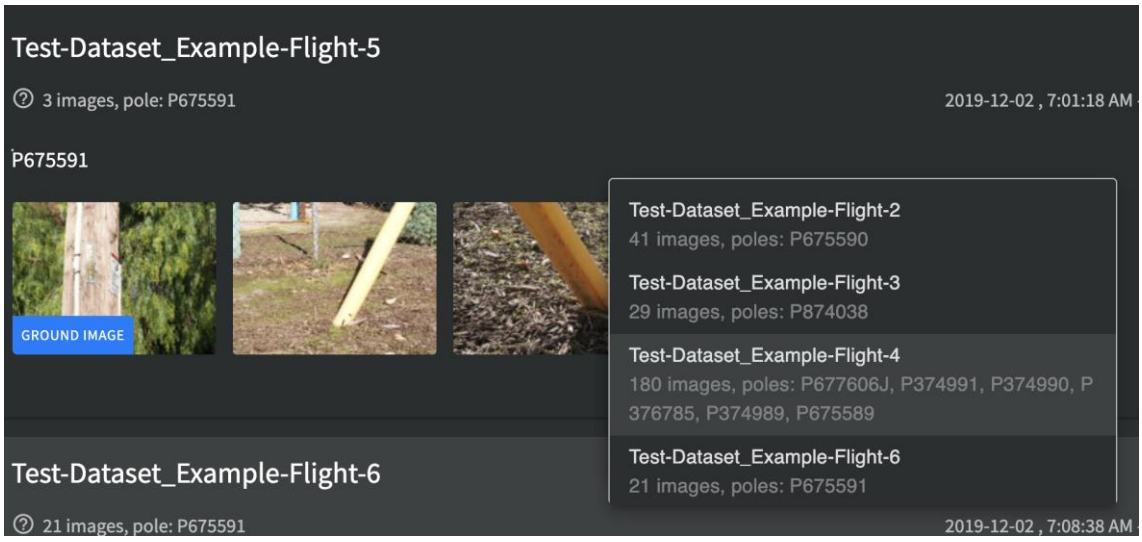

Figure 27. Image move to submenu of valid datasets.

Clicking on the View EXIF and XMP metadata opens a scrollable modal with the image's metadata in a JSON format as seen in the figure below. This could be useful to quickly review the image's original metadata.

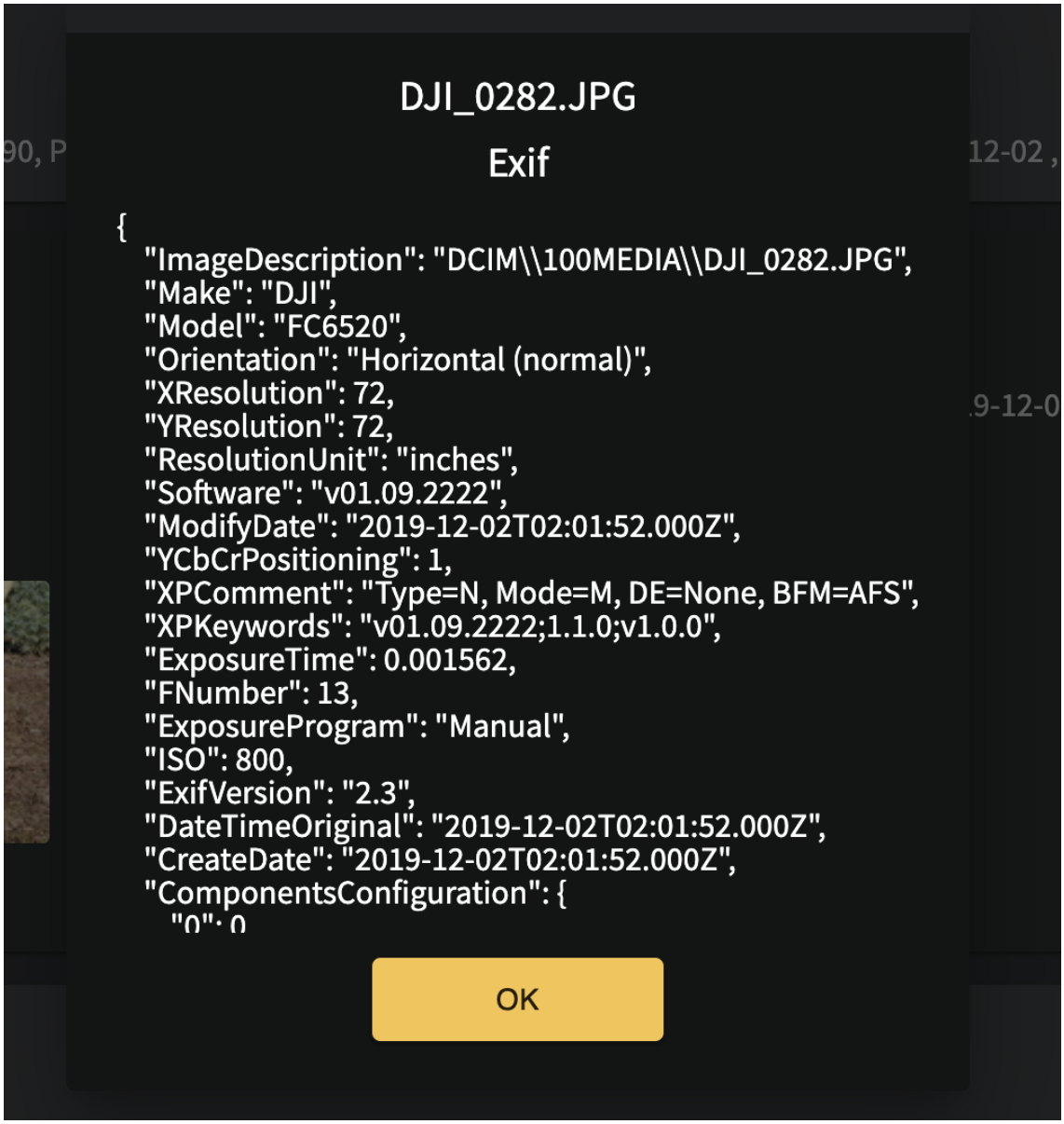

Figure 28. Image metadata modal.

To select multiple images the user could either click them one by one or click and drag to select a few in one go. This opens the image toolbar at the top of the application as seen in Figure 29.

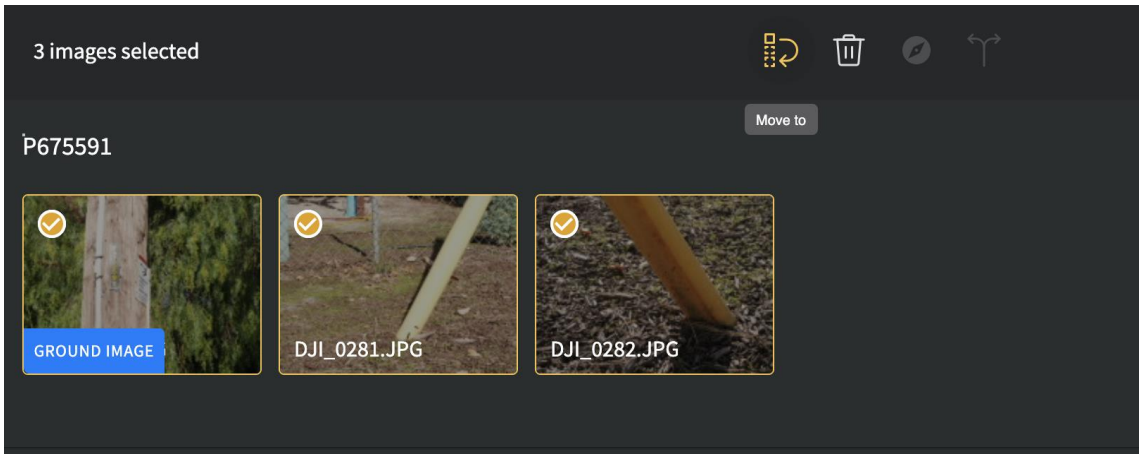

Figure 29. Multi Selection and image toolbar.

One specific feature that can be accessed only from the image toolbar is the ability to split a dataset into two datasets at a specific image. For this button to be enabled, exactly one image must be selected. This image will be the first image in the new dataset, and that new dataset will also contain all images after the selected image.

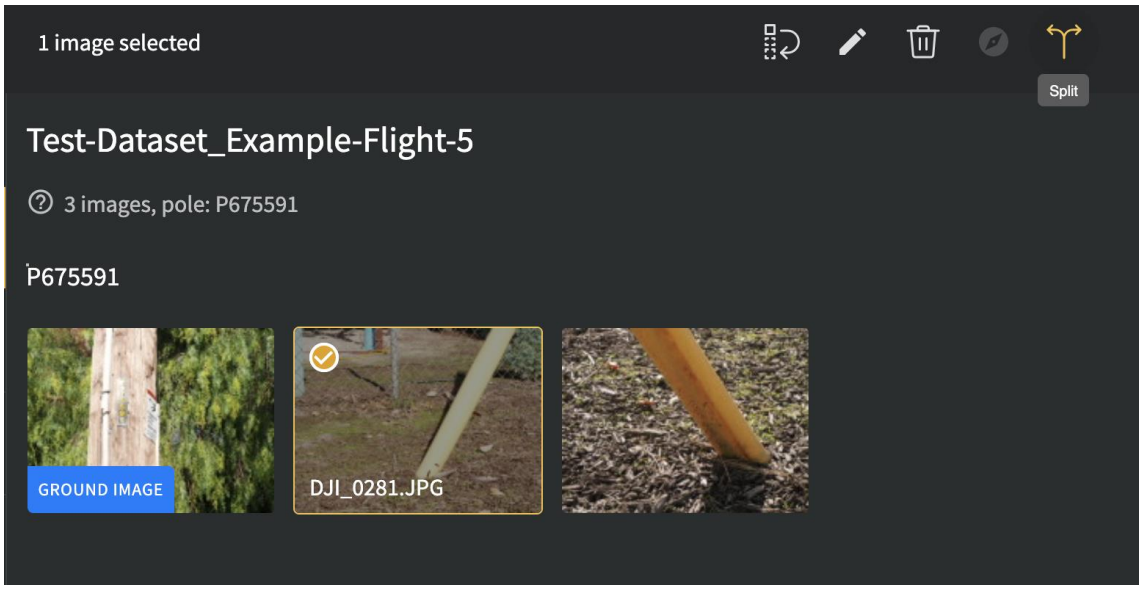

Figure 30. Split a dataset into two datasets with the image toolbar Split option.

Clicking on any image opens it up in full resolution in the Lightbox preview, where images can be navigated, as well as zoomed in and out and panned around. This feature is very helpful for quickly inspecting a specific detail.

Figure 31. Image light box modal.

5.4 Image to Asset Assignment

Clicking the “Next” button continues the application’s flow into the next step of the wizard. At this point it is possible to assign images to assets via the map. The map renders pins for each image, including a depiction for the image’s orientation. In addition to that, pins for each asset in the area are also rendered.

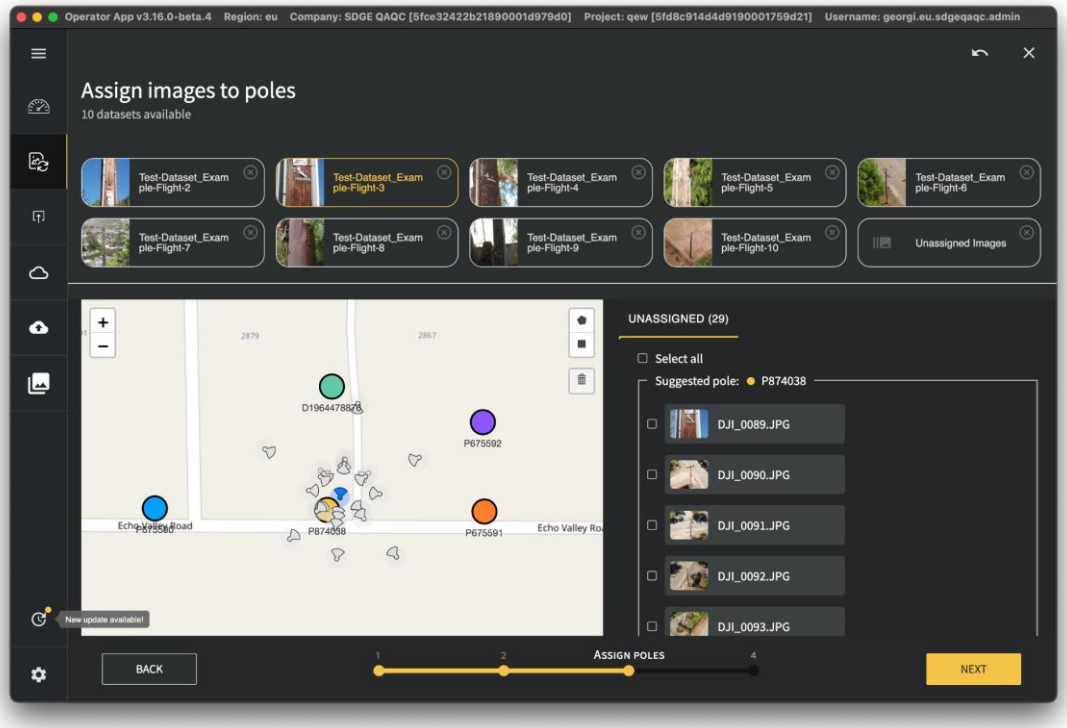

Figure 32. Image assignment map.

To assign images to assets first a selection must be made. That can be accomplished in a few different ways. It could be done by either clicking on each image's checkbox individually, or by clicking the "Select all" checkbox to select all images in one go. Multiple images could also be selected by clicking and dragging. Finally, another way to select images is by using the map tools to draw a rectangle or a free form polygon and that will select all images within its bounds.

Figure 33. Selecting images on the map.

The image assignment toolbar UI element appears at the top of the application as soon as a selection has been made. The toolbar allows for custom searches of an asset by ID, as well as to select from the listed assets. Additionally, the assignment toolbar also maintains a list of suggested assets that should be the best fit for the currently selected images. Alternatively, a quick and easy way to assign images to assets is to just click an asset on the map while having images selected.

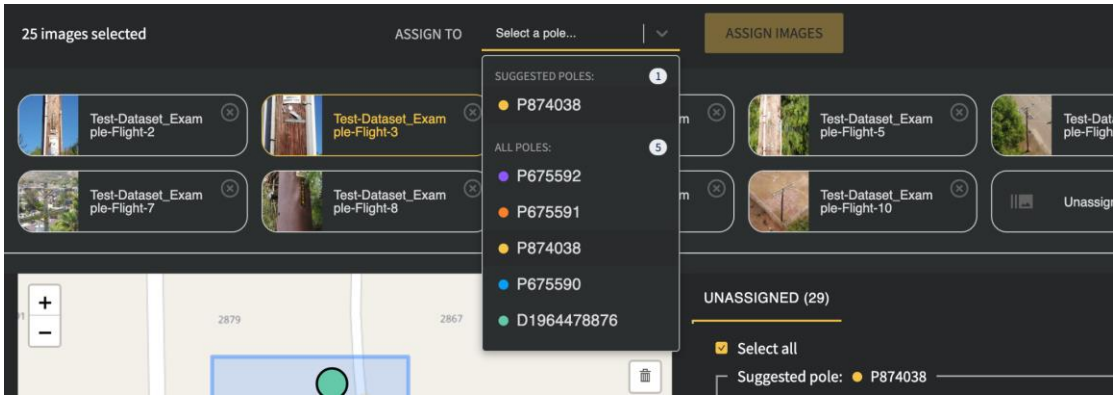

Figure 34. Assign images to assets dropdown.

When all images from that dataset are assigned, they could be found under their corresponding asset tab and the main tab where all the unassigned images were earlier, will let the user know that all images are now assigned.

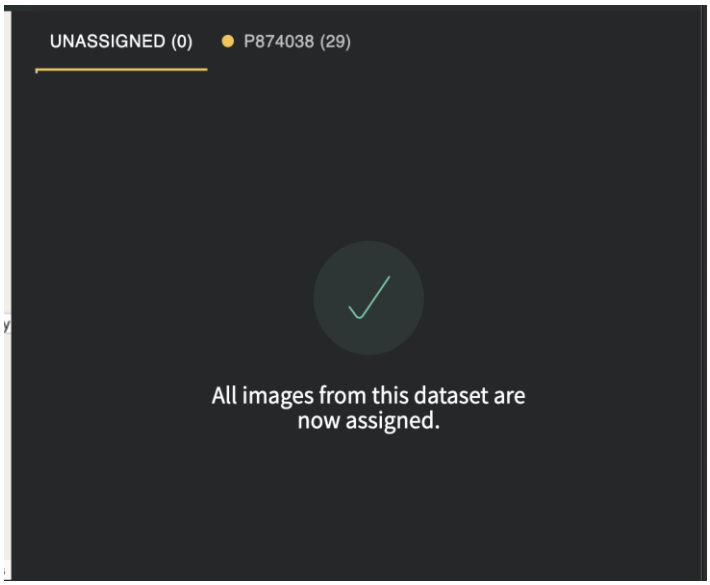

Figure 35. All images have been assigned to assets.

5.5 Pre-Export Feedback and Exporting Datasets

Continuing the application flow and reaching the next step of the wizard displays a list of the datasets to be exported. This list highlights if the datasets are in good standing or if there are any potential issues that might prevent a successful export. Hovering over the red question mark element would provide more information as to what might be wrong or be missing. Most issues should be actionable, meaning if the user forgot to assign some images to assets, they can go back a step, do that, and come back to the export view when ready.

The export will generate all the necessary dataset files for what is considered to be a dataset for Sharper Shape and copy over the images and pdf's part of those datasets to a specific folder structure.

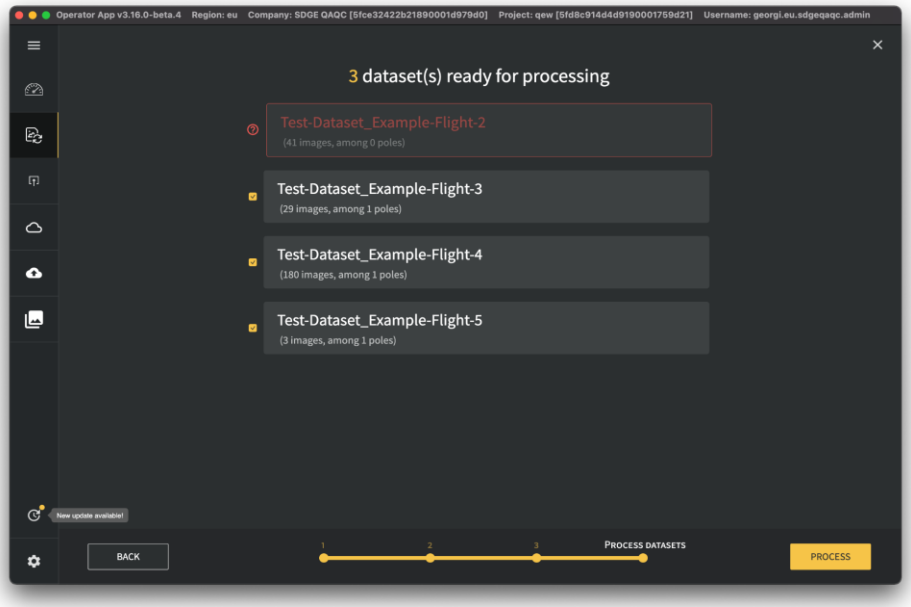

Figure 36. Export datasets view.

Figure 36 provides an example of the export dataset view when one dataset has some issues, as noted by the colour of the first dataset in the list being red.

Finally, Figure 37 below shows the “export successful” view.

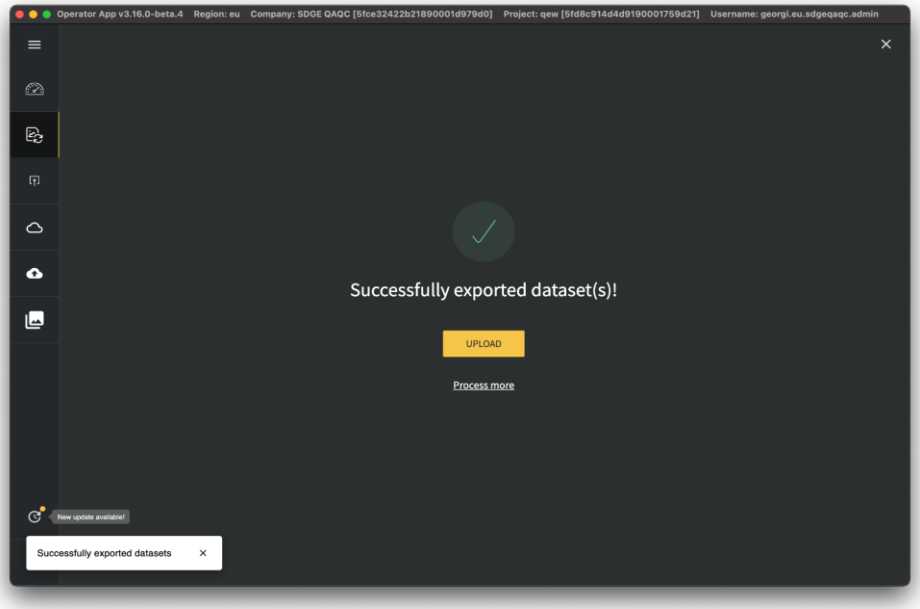

Figure 37. Datasets export successful.

5.6 One Step Tool – Alternative, Simpler UI

While the original Dataset Generator module is robust and configurable, it does require going through a number of steps and tweaking and configuring the setup in order to export datasets. A bit further into the project's development a need was identified for an easier way to produce and export datasets even quick. Enter "One Step Tool" which, as the name aptly suggests, aims to help export datasets in just one step.

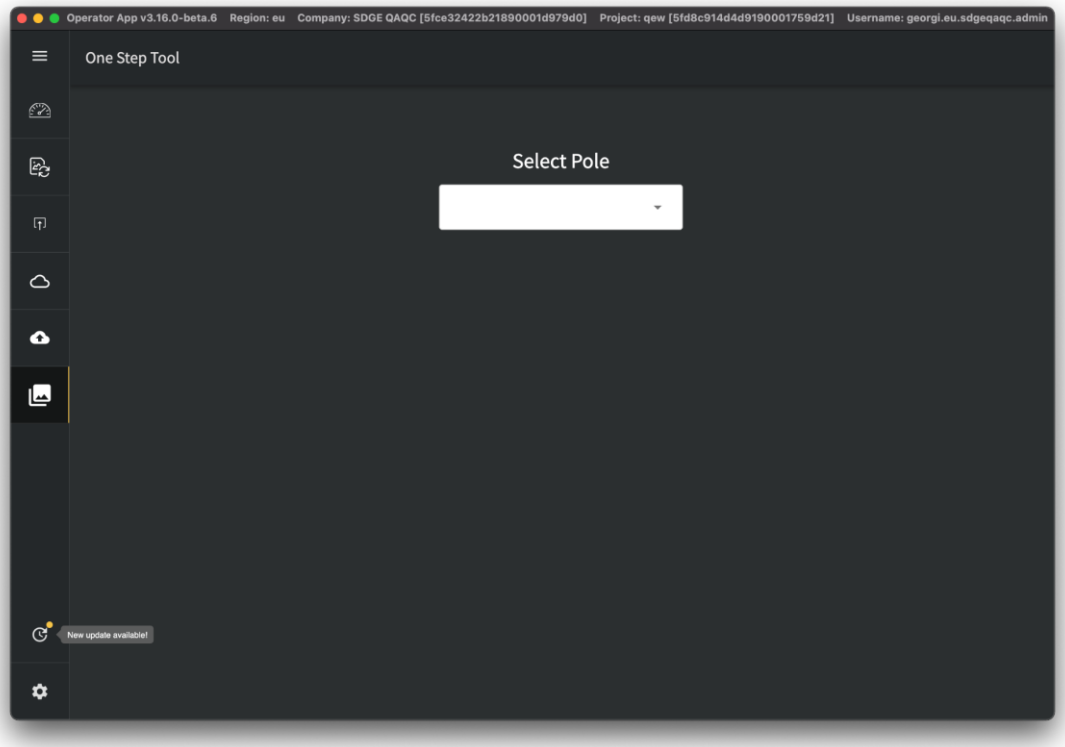

Figure 38. One Step Tool.

The One Step Tool (for a lack of a better name) was the attempt to speed up and streamline the dataset creation process if 1 important assumption could be made - *The user knows the pole / asset they are uploading the images to*.

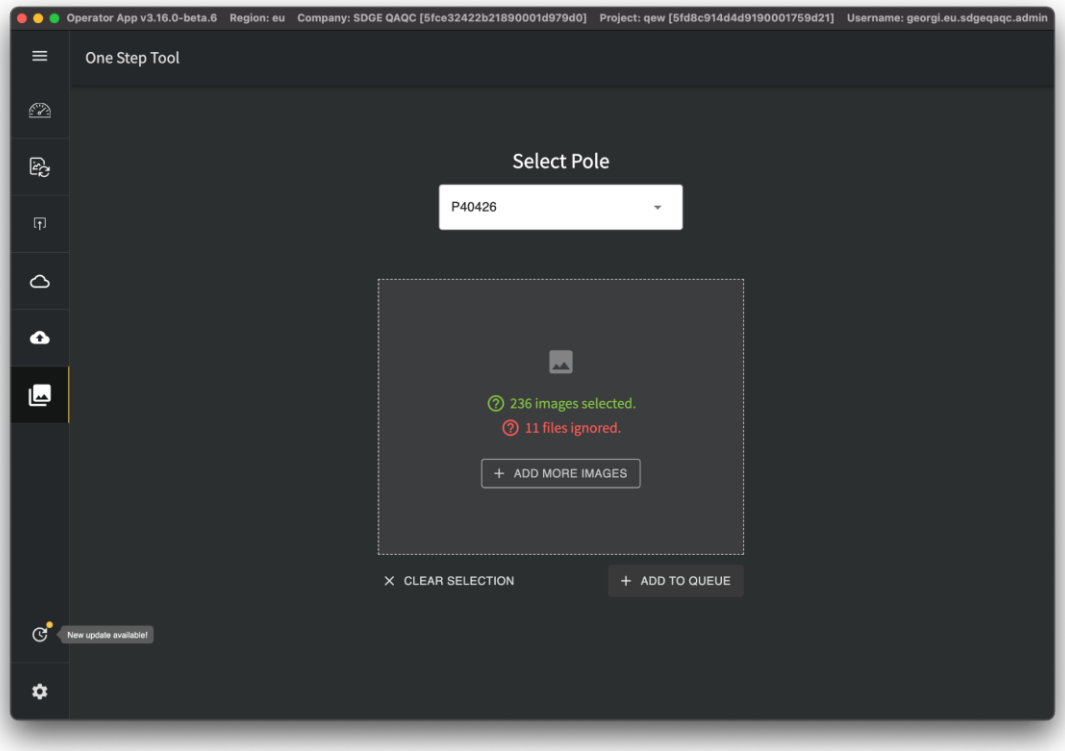

Figure 39. One Step Tool with a pole selected and images dropped.

In this case the dataset creation flow starts by first selecting the asset to assign the images to and this reduces the necessary number of steps and simplifies the dataset creation process. Under the hood, the One Step Tool uses the exact same dataset processing functions, as the Dataset Generator module, and this code reusability has made it easy and fast to ship this alternative UI flow for dataset exports.

Once the asset is selected and some images dropped, the user can click the “Add to queue” button which puts this potential dataset to the export queue. The flow can be repeated for as many assets and images as required and when done, clicking the “Process” button at the top right exports all datasets.

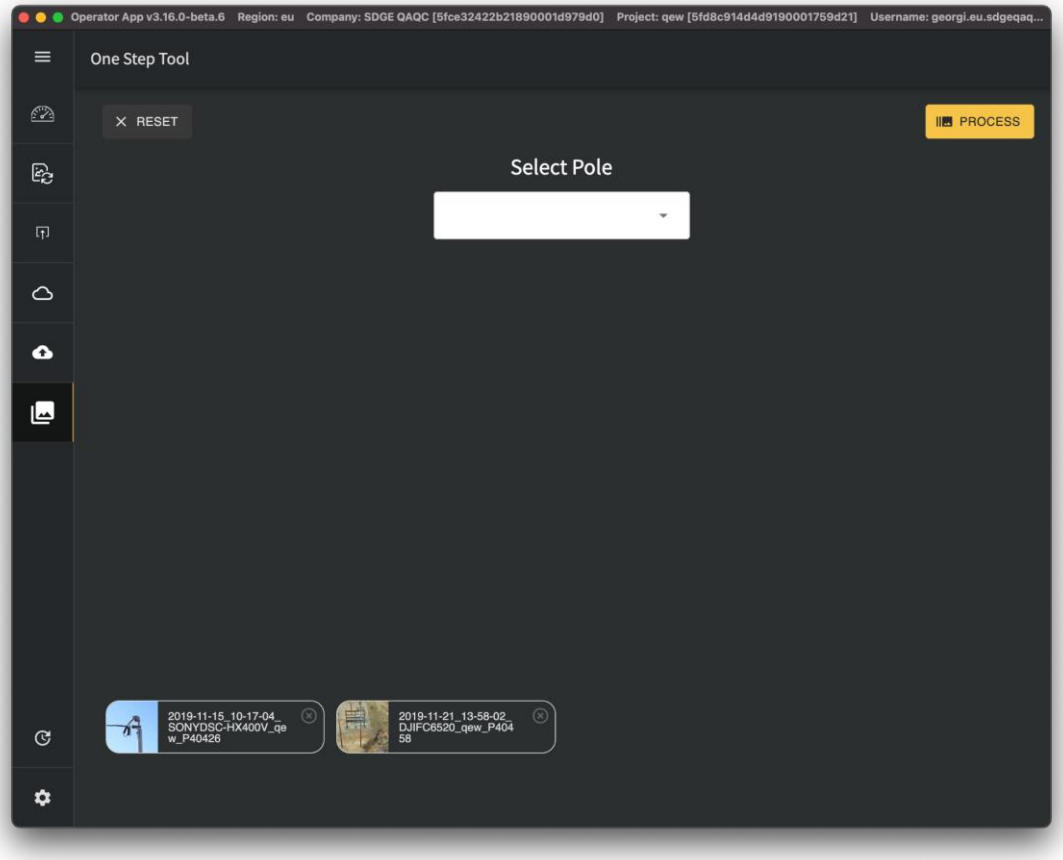

Figure 40. Datasets in the export queue, ready to be exported from the One Step Tool.

6 Discussion and Conclusions

Work on this project has undergone many iterations and has been with a heavy focus on Agile methodologies. Early feedback, as well as constant on-going feedback have been critical to the project's short-term success. All initial goals have been accomplished and additional features and quality of life improvements have been added along the way. That being said, there is a lot more to come, in terms of future development.

6.1 Project Feedback

Feedback from stakeholders has been critical to this project. Because the OA deals with user data in the form of images, especially early on it has been very beneficial to receive different types of data, as different images and different formats have highlighted potential issues and corner cases the app had to cover, to provide a good user experience, and allow dataset creation while at the same time making sure data integrity and quality is preserved. This had been one of the most enjoyable aspects of the work, since our drone crews, US image processing folks, as well as Finland based drone test pilots have been very helpful and provided the necessary data for testing, as well as feedback on issues, which were then handled quickly in each release.

Additionally, having shipped the application beta releases feature early in the process, allowed for releasing "less polished" beta versions of the application with new features, which were then first tested internally, or by anyone who opted in for beta releases. This alone allowed to ship new features and at the same time preserve a roughly 2–3-week stable release cycle.

6.2 Future Improvements

Regarding future improvements, there is no shortage of good ideas and improvements for the application. While the main functionality is in place and covers a plethora of use cases and images by different drone vendors, there is

always room to improve. One big topic is always on the performance side of things. The application currently processes images decently fast, even when it comes down to processing thousands of images at a time. Exporting itself also seems to be working decently fast. However, as software engineering is such a fast-paced field and new patterns and ideas emerge, it is nearly for certain that performance can always be optimized further.

One example of a quality-of-life UI improvement would be to display release notes in a message box right in the application after an update. Currently these notes are delivered via other channels. Another item for the future is to consolidate image labels. Those labels were something that was designed early on, but as time progressed, more and more labels were needed to highlight different image statuses, such as blurry image, duplicate image and so on. Sometimes this information is communicated via a border with a specific colour, other times via a label. We can do better and the way to communicate issues should be consolidated, made apparent and clear to the user.

Another example is around fetching hundreds of thousands of assets information and rendering those pins on the map, which is sometimes necessary when dealing with large areas and a lot of images. Perhaps the way the map renders items could be re-considered and re-written, such that the logic is based on rendering items only visible on the viewport. There are pros and cons either way and a lot of factors to consider when making these decisions.

Lastly, constant, and on-going feedback from end users is something we have appreciated a lot and are planning to always be on the lookout for, since it has led to solving a lot of problems and improving the application in many ways. Continuing to listen to users and acting on their feedback is one of the biggest future improvements topics, because at the end of the day, the application is there to serve its users and make tasks easier to handle.

References

1 ElectronJS project. https://www.electronjs.org/ Accessed 19 Mar 2022.

2 React. https://www.reactjs.org/ Accessed 19 Mar 2022.

3 TypeScript. https://www.typescriptlang.org/ Accessed 19 Mar 2022.

4 Node.js. https://nodejs.org/en/ Accessed 19 Mar 2022.

5 Sharp. https://sharp.pixelplumbing.com/ Accessed 19 Mar 2022.

6 Exifr. https://github.com/MikeKovarik/exifr Accessed 19 Mar 2022.

7 Leaflet. https://leafletjs.com/SlavaUkraini/ Accessed 19 Mar 2022.

8 Turf. https://turfjs.org/ Accessed 19 Mar 2022.

9 Lloret Romero, N., Gimenez Chornet, V.V.G.C., Serrano Cobos, J., Selles Carot, A.A.S.C., Canet Centellas, F. and Cabrera Mendez, M. (2008), "Recovery of descriptive information in images from digital libraries by means of EXIF metadata", Library Hi Tech, Vol. 26 No. 2, pp. 302-315.

10 Martin Harran, William Farrelly, Kevin Curran, A method for verifying integrity & authenticating digital media, Applied Computing and Informatics, Volume 14, Issue 2, 2018, Pages 145-158, ISSN 2210-8327

11 EXIF tag names list. https://exiftool.org/TagNames/EXIF.html Accessed 20 Mar 2022.

12 XMP tag names list. https://exiftool.org/TagNames/XMP.html Accessed 20 Mar 2022.

13 Lerna. https://lerna.js.org/ Accessed 20 Mar 2022.