# Stack Pipeline CNN

DOCUMENTATION

Rau, Moritz

LUFTBILD UMWELT PLANUNG

August 15, 2023

# Contents

This documentation is a variation of the arc42 template. See https://arc42.org.

# 1 Introduction and Goals

This pipeline is designed to process Digital Terrain Model (DTM), Digital Surface Model (DSM), normalized DSM (nDSM), and orthophoto (ORTHO) tiles of varying formats. Its primary function is to generate a stack of nDSM and ORTHO from the input data. The pipeline is engineered to efficiently handle multiple datasets within a single processing instance. Its primary purpose is to provide data in a specified format for a Convolutional Neural Network (CNN) model. Concurrently, the pipeline generates mosaics as byproducts. To enhance user accessibility, a graphical user interface (GUI) has been developed.

Primary objective was to simplify and automate preprocessing tasks as much as possible. This involves creating a pipeline that seamlessly manages new data, replacing manual processing steps and custom code adjustments with a consistent and standardized method.

## 1.1 Requirements Overview

- Conversion of various file formats into TIF, ensuring data integrity and completeness
- Mosaicing of TIF tiles
- Effective handling of mosaic and tile inputs
- Generation of nDSM based on DTM and DSM
- Stacking ORTHO and nDSM, including raster alignment and resampling
- Production of output in a specific format required by the model

## 1.2 Quality Goals

These functions should be implemented with specific considerations:

- Functionality: The pipeline must cover diverse cases, including various file formats.
- Maintainability: The software's modularity should enable the extraction or replacement of specific modules for flexibility.
- Performance: Performance is not the primary focus due to limited expertise, but optimization is essential given the time-intensive processing nature.

## 1.3 Stakeholders

- Creator: Moritz Rau
- LUP Python Developers:
  - Benjamin Stöckigt, Sebastian Lehmler, and others
  - Access to architecture for further development
  - Pipeline is accessed through: GUI, script
  - Setting up tool for normal users
- LUP normal user:
  - Access to README
  - Pipeline is accessed through: GUI
  - No access to architecture

# 2 Architecture Constraints

The pipeline must operate on a single machine. As a result, hardware limitations, particularly concerning memory and storage, must be taken into account. The programming language utilized is Python. The system's ease of migration to leverage existing resources is a priority. Moreover, the design should facilitate maintenance by individuals not involved in its initial creation. Since the creator might not be easily reachable, a straightforward and uncomplicated approach is favored.

# 3 System Scope and Context

## 3.1 Business Context

This tool is the outcome of a summer 2023 internship (Moritz Rau, B.Sc. Environmental Sc.). Its main objective was to create a pipeline for generating stacks in a specified format to serve as input data for a CNN model (Sebastian Lehmler, Benjamin Stöckigt). Over time, the pipeline's functionalities expanded to accommodate various exceptions and deviations from a simple stacking process. To achieve a more systematic and user-friendly implementation, the pipeline was thus integrated into a more complex architecture which is described in this document.

As part of this effort, a GUI (Graphical User Interface) was developed to make the tool accessible to non-Python users. However, it should be noted that the GUI received lower priority during implementation and, as a result, may not be as refined as other aspects of the tool.

Initially designed for Sebastian Lehmler and Benjamin Stöckigt only, efforts have been made to make the software accessible to more LUP workers.

Detailed documentation is provided to support maintenance and further development.

# 4 Solution Strategy

The fundamental concept revolves around maintaining a repository that holds all important files and information. This concept makes it easy to install the pipeline on different machines by simply copying the repository. This repository serves as a central space for storing scripts, documentation, and establishing a structured folder system for temporary data within the automated process. This organized structure becomes the foundation upon which the processing logic relies. It's worth noting that input and output files are not restricted to this repository location; they can reside anywhere accessible to the machine. This system architecture is visualized in figure 2.

At the core of this repository is the pipeline.py file, which defines the processing logic. The activity diagram (figure 1) provides a detailed abstract view of this processing logic. The intricate steps of individual processing are primarily defined in processing_functions.py. This separation of the "what" from the "how" lends modularity to the pipeline, simplifying maintenance and enhancing readability.

To execute the pipeline, only specific inputs are needed—primarily paths to raster files and related metadata (such as city, year, etc.). While a direct approach could involve embedding file paths within the script, a different approach was chosen. This decision stems from the necessity to manage multiple pieces of related data for each dataset (associated with a stack). Particularly when processing multiple datasets in batches, managing all this data directly within the script's beginning becomes complex. To address this, a deliberate effort was made to create separate files that aptly organize this data. Recognizing the potential for errors and the inconvenience of manual file creation, a GUI was implemented to generate these files. Acknowledging the GUI's limitations, a less user-friendly but more robust function is also available. Further development can also be applied to this function.

The pipeline's structure is built on a well-defined folder organization, encompassing Input, Temp, and Output directories.

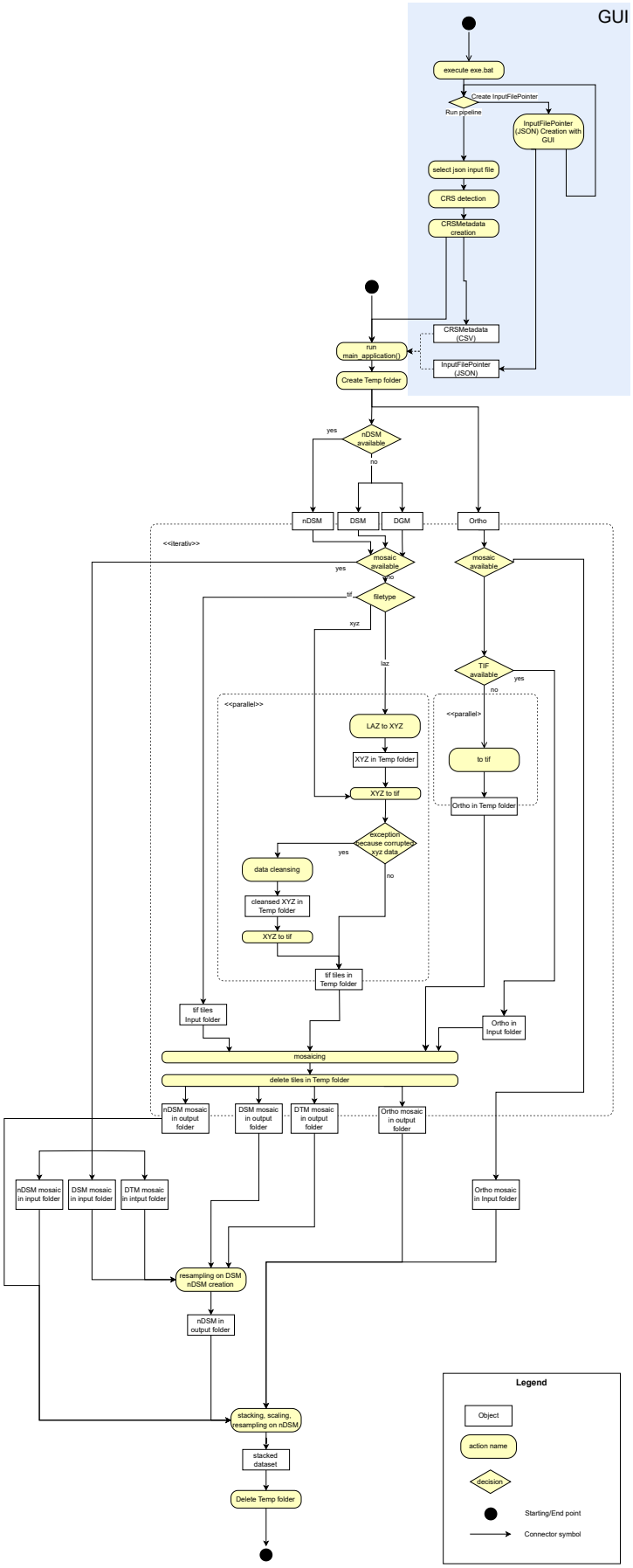



Figure 1: Activity diagram

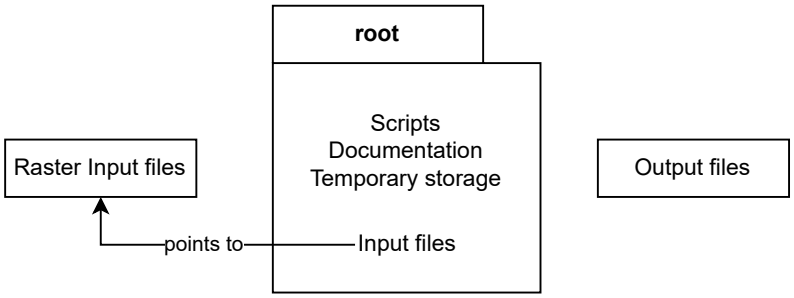



Figure 2: Repository architecture. The root folder is the repository

The Input folder holds essential pipeline input files, while the Output folder offers a structured system for output files. The Temp directory serves as a workspace for interim steps. This architectural layout enhances clarity, making comprehension and management more straightforward.

Striving for a balance between extensive automation and user control to ensure data quality was one of the key issues. Error-handling mechanisms were implemented to bypass potentially flawed data. A cleansing step was introduced to make as much data available to the pipeline as possible. A logger module was established to provide a comprehensive overview of the entire process giving the end user information if somthing went wrong. To minimize user-defined parameters, certain decisions—such as resampling mechanisms—were hard-coded and optimized for the CNN model input. This is worth noting when the pipeline is used for different purposes.

Other key decisions are:

- **Stack extent:** The stack's boundaries are set by the nDSM bounding box. Generally it's possible that in certain locations, some bands may have data while others do not.
- **Dataset Identifier:** To differentiate datasets during processing, we adopted a identifier consisting of 3 keys. This includes the city name, year, and an optional descriptor, forming a unique identifier. The descriptor can be chosen freely to better describe the dataset, such as indicating the season (winter/summer) or other relevant attributes. While it's recommended to use meaningful strings for all three components, there are no strict checks imposed, allowing flexibility in naming choices.
- **Naming for Stacks and Mosaics:** Our automized naming strategy involves incorporating dataset keys when naming stacks. In contrast, mosaics are named based on metadata stored with the DSM, DTM, nDSM, or ORTHO, only considering the city name of the key.
- **Nodata dictionary** There is a pyhton dictionary stored in default_nodata_dict.py which is imported into the pipeline and the processing functions. It's only purpose is to avoid hardcoding nodata values. For every data type there is one defined nodata value. If this value should be changed only the dict needs to be changed.
- **Overviews and Compression** Upon the completion of each final product (mosaics and stacks), the file undergoes compression, and overview layers are generated. This enhances the ease of opening these files in a viewer. It's worth noting that due to the incompatibility of BIGTIFF and compression in geowombat, achieving compression often necessitates an extra step.
- **GDAL Usage Approach:** From a programming perspective, we exclusively employ the command-line version of GDAL through subprocess, not making use of GDAL's Python API. This deliberate choice ensures the robustness of the pipeline by addressing potential compatibility concerns that might arise with the Python API. (`https://rasterio.readthedocs.io/en/latest/topics/switch.html`)
- **Parallel Processing with Python's Multiprocessing:** Whenever possible, we embrace the utilization of Python's multiprocessing library to enable parallel processing. This decision greatly enhances the performance of the pipeline by allowing multiple tasks to be executed simultaneously, thereby optimizing overall processing speed.
- **Advanced Raster Tasks with Geowombat:** For intricate tasks like raster alignment, we turn to the geowombat library. This library stands out due to its integration of dask's parallel computing capabilities.This incorporation expedites processing, resulting in improved performance and simplified maintenance.

- **Setting up Virtual Environment with Anaconda or Mamba:** To begin using our solution, it's essential to establish the required virtual environment using either Anaconda or Mamba on the machine. Detailed instructions for this setup process are comprehensively outlined in the README file, providing clear guidance for users.

#### Functional Overview

Table 1 is presenting an overview over the technical specifications of the pipeline. Users should allways be aware of what happens inside the pipeline. Please keep in mind that changes might have happend after this report was finalized. Check out if there is further documentation. If you update the pipeline please alter the README accordingly.

# 5 Building Block View

## 5.1 Pipeline (pipeline.py)

The script begins with a configuration section at the top, followed by the main function, and concludes with the execution of this main function. To manually execute the pipeline without utilizing the GUI, one simply needs to configure the settings in the configuration section and subsequently run the script. When using the GUI, the configuration section is disregarded, as the function parameters are furnished by the GUI. In such cases, the GUI imports the main function.

Proper identifiers (city, year, description) are crucial for a succesful execution. In cases where the JSON is created manually, special attention must be given to ensure proper identifier usage and correct looping through the dataset.

### 5.1.1 Error handling

We implemented a try except mechanism. In broad terms, our approach is as follows: if tile conversion encounters an issue, the respective tile is skipped and the occurrence is logged. If mosaicing, nDSM generation, or stacking encounters errors, the entire dataset is skipped and a corresponding entry is logged. For functions using subprocess we spend some additional effort to integrate them into the error handling mechanism.

### 5.1.2 File extraction

To make the pipeline as flexible as possible we implemented a file detection module which works like this: The provided input folder for DTM, DSM, nDSM or ORTHO and all its subfolders are scanned for any kind of file. The most frequent filetype is considered as the relevant one. All files of this type are recognized as tiles for subsequent processing. Files of other types are disregarded in the subsequent steps. The purpose behind this is to prevent potential complications arising from metadata files like .txt, which else would be considered as rasterfiles and undergo processing with GDAL if not properly filtered out. In general it is important to not mix data from different datasets. Currently supported fileformats are ".xyz", ".txt", ".las", ".laz", ".jp2", ".jpg". Check the code for updates.

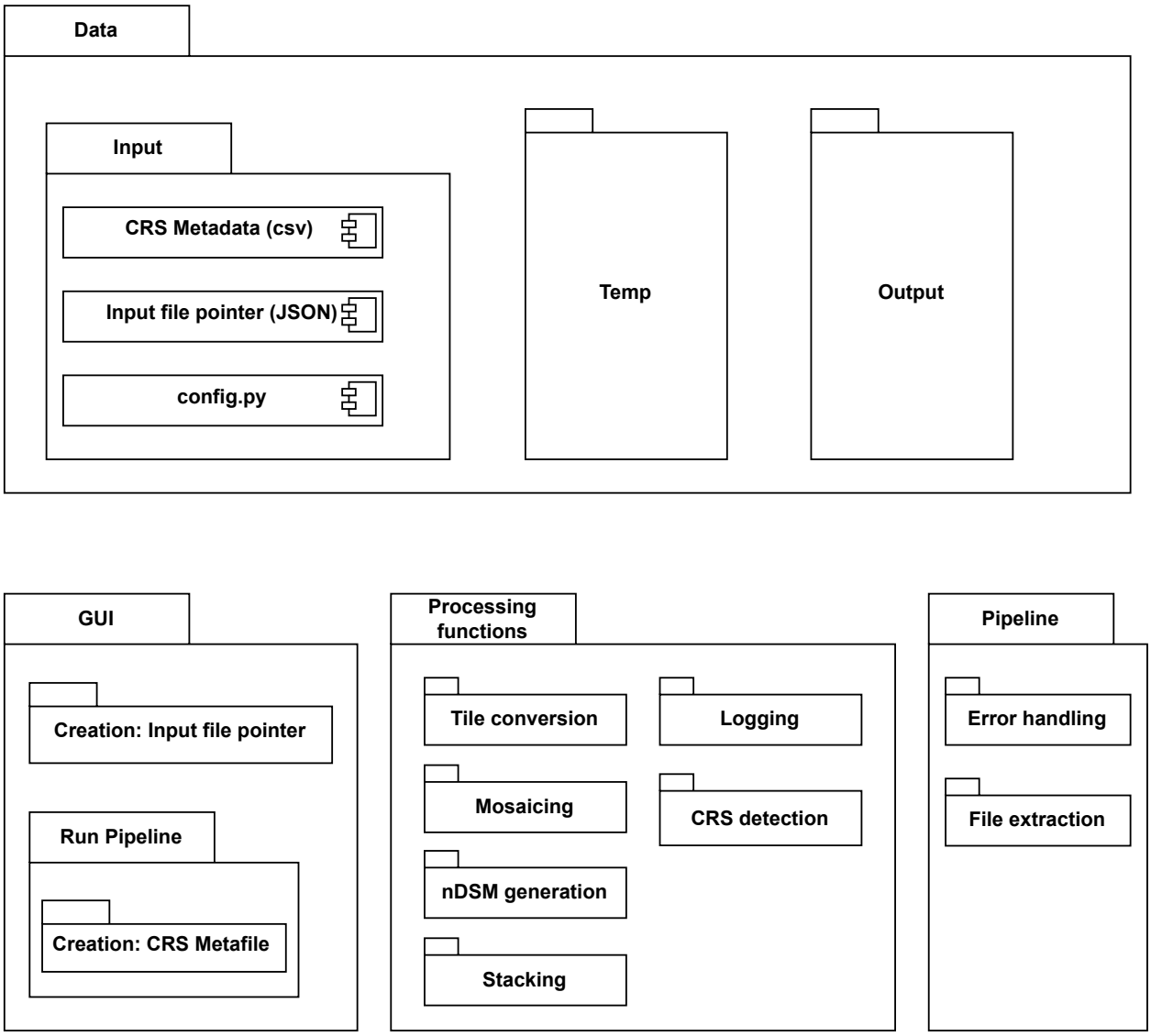
## 5.2 Processing functions

### 5.2.1 Tile conversion

The objective of tile conversion is to transform various file types into TIF files. For formats like XYZ, JPG, and JP2, we employ GDAL WARP.

| **Function Name** | **dType (out)** | **Nodata (out)** | **CRS** | **Resampling** | **Compression** | **Resolution (/reference for alignment)** |
|---|---|---|---|---|---|---|
| PDAL XYZ extraction | any | | | | | |
| GDAL WARP (XYZ to TIF) | detected input dtype | f(detected_dtype) | CRSMetadata (csv) | | | |
| GDAL WARP (XYZ (las) to TIF) | float32 | f(float32) | CRSMetadata (csv) | | | |
| GDAL WARP mosaicing | any | any | CRSMetadata (csv) | nearest | LZW | if <20cm = 20cm |
| Geowombat nDSM creation | float32 | f(float32) | DSM | DTM : nearest | LZW | DSM (!could be smaller than 20cm) / DSM |
| Geowombat stack | uINT16 | f(uINT16) | nDSM | ORTHO : nearest; nDSM : upsampling: cubic, downsampling :nearest | LZW | config.py /nDSM |

Table 1: Technical specifications




(a) Package diagram

| Name | Responsibility |
|---|---|
| Pipeline | Defines the processing flow of the system. Allows manual execution of the pipeline. |
| Processing Functions | Contains all the functions required for data processing. |
| Data | Provides a structured approach to manage Input, Temporary, and Output data. |
| GUI | Enables the creation of Input data files and serves as a starting point for the pipeline. |

(b) Responsibilities in the System

Figure 3: Package diagram and the funciton each package has

LAZ/LAS files are not directly converted to TIF. Instead, we extract XYZ coordinates from them for further processing with GDAL. To accomplish this, we rely on PDAL. This approach is taken because LAS/LAZ data from German municipalities often consists of regularly gridded, processed data points rather than actual LIDAR data. Consequently, only this specific type of LAS/LAZ data is suitable for processing within the pipeline. Converting these files to XYZ format only reveals the true nature of the data structure.

Due to GDAL's strict data format requirements (sorted and grid-based), we've implemented a try-except logic that includes data cleansing. The strategy involves always attempting direct GDAL processing first. If an error arises, the XYZ data undergoes cleansing by filtering out off-grid data points and sorting them. A tolerance of 4mm is applied, as this error is typically considered an artifact rather than true off-grid data. The core process of cleansing revolves around extracting the probable source and potential resolution. Utilizing this information, data points that do not fall within 4 mm of a virtual grid, defined by the origin and resolution, are filtered out. The cleansing process is only employed when necessary to enhance performance. In practice, this simple cleansing method renders a substantial amount of data usable.

Tiles that have been converted and/or cleansed are stored in the Temporary folder as they are slated for deletion once the mosaic is generated. Tiles already in TIF format remain in their original locations. The mosaicing process will directly reference this specific location.

The nodata value specified in the JSON that defines the input nodata for the tiles is taken into account during the file conversion process. If not defined GDAL will try to read the nodata from the header. If a value is provided in the JSON the header nodata is neglected.

To handle a batch of tiles efficiently, we harness the multiprocessing library, enabling parallel processing for improved speed and efficiency.

### 5.2.2 Mosaicing

This step is skipped if a mosaic is provided directly. Mosaics keep either there resolution or get downscaled to 20cm if the resolution is smaller than 20cm. We utilize GDAL WARP for this task. Even if GDAL WARP can handle a variety of filetypes this function should be used with TIF only (nodata, CRS). However, due to the use of CMD, there's a character limitation for the command ( 32,000 characters)(https://learn.microsoft.com/de-de/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa?redirectedfrom=MSDN). To overcome this limitation, we break down the implementation by dividing tiles into groups, each forming a sub-mosaic that's temporarily stored. These sub-mosaics are then combined in a final step to create the complete mosaic.

### 5.2.3 nDSM generation

For nDSM generation is carried out using geowombat to facilitate alignment and resampling conveniently. The process involves straightforward subtraction of the DTM from the DSM. Opting for the DSM as the alignment and resolution reference is deliberate, given its higher relevance to the final product, and to minimize resampling whenever feasible. Data points that fall below 0 are adjusted to 0.

### 5.2.4 Stacking

Stack creation is carried out using geowombat. The nDSM is utilized as the alignment reference and any nDSM values below 0 (as in the nDSM generation step) are adjusted to 0. This accommodation is made in situations where the nDSM is externally generated rather than within the pipeline. Both ORTHO and nDSM are scaled by a factor of 256 and stored as UINT16, as this format is mandated by the CNN model. In very rare cases this can lead to information loss if high buildings above 256m are available. For the ORTHO data, the interpolation method is set to nearest neighbour while for nDSM data, upsampling employs cubic interpolation, and downsampling utilizes the nearest neighbour method. The consistent band sequence is as follows: ORTHO is positioned first, followed by nDSM.

### 5.2.5 Logging

Logging plays a crucial role as the pipeline is designed to process multiple datasets, a task that can span over several days. Constant monitoring of the CMD is not practically feasible in such scenarios. Moreover, the pipeline is intended to continue running even if errors occur. These errors, however, should still be documented in a log file and presented to the user upon completion.

To accomplish this, two files are employed:

The ”pipeline.log” file serves as a repository for all logs in their order of occurrence. This encompasses errors, execution times, and more. Logs from previous pipeline runs are retained, allowing users to trace back issues if they arise. While this file might appear cluttered and challenging to read, it primarily functions as a means of issue tracking. It contains all error messages to make debugging easier if needed. Important logs are denoted with either #failedtile or #failedcity.

The ”Overview.log” file is the resource that every user should consult at the conclusion of the pipeline process. It furnishes an overview of the runtime, aiding in the identification of time-intensive steps. Additionally, it offers a list of failed datasets and tiles, along with reporting the count of warnings. It’s recommended to first review this file and delve deeper into the ”pipeline.log” if needed.

### 5.2.6 CRS detection

The process of detecting the Coordinate Reference System (CRS) is conducted either for a list of files pertaining to a dataset or for an individual file that constitutes a mosaic. In the case of all file formats excluding LAS/LAZ, a maximum of 10 files are randomly selected. The EPSG code is then extracted using rasterio for the chosen files. If all these files share the same EPSG code, it is assumed to be the valid CRS for the dataset. For LAZ/LAS data, up to 3 files are checked. As soon as one provides an EPSG code this one is assumed valid. This is because PDAL’s processing is comparably slower compared to rasterio.

## 5.3 GUI

The GUI has been created only for convenience reasons while developing at the beginning. Thus it’s mainly created by using CHATGPT. During the development at became clear that there is a demand for the GUI making it possible for normal non-pyhton users the use the pipeline. Some effort has then be made to make the GUI more reliable , still it is only a small sideproject within the whole project and not super robust. One can use it but further development of the GUI is not reccomended.

It provides the functionalty to create both required input files (InputFilePointer, CRSMetadata) in an easy and convenient way. Furthermore it makes use of the CRS detection module while creating the CRSMetadata file. Normale users can run the GUI through a batch file and because it is an entering point for the pipeline there is no need to run a python script manually anymore.

## 5.4 Input

### 5.4.1 InputFilePointer Json

The InputFilePointer file stores all the paths to files and folders (+Metadat) which are required to build the stack. The prefered location to store it is: root ¿¿ data ¿¿ InputFilePointer. The JSON file’s data model is visually depicted in Figure 4. The keys ”city,” ”year,” and ”description” function as unique identifiers for each dataset. It is advisable to employ meaningful strings for these keys since they also serve as the basis for naming the stack. However, there are no strict formatting requirements for these keys. It’s worth noting that a key can even be an empty string. To ensure data integrity,

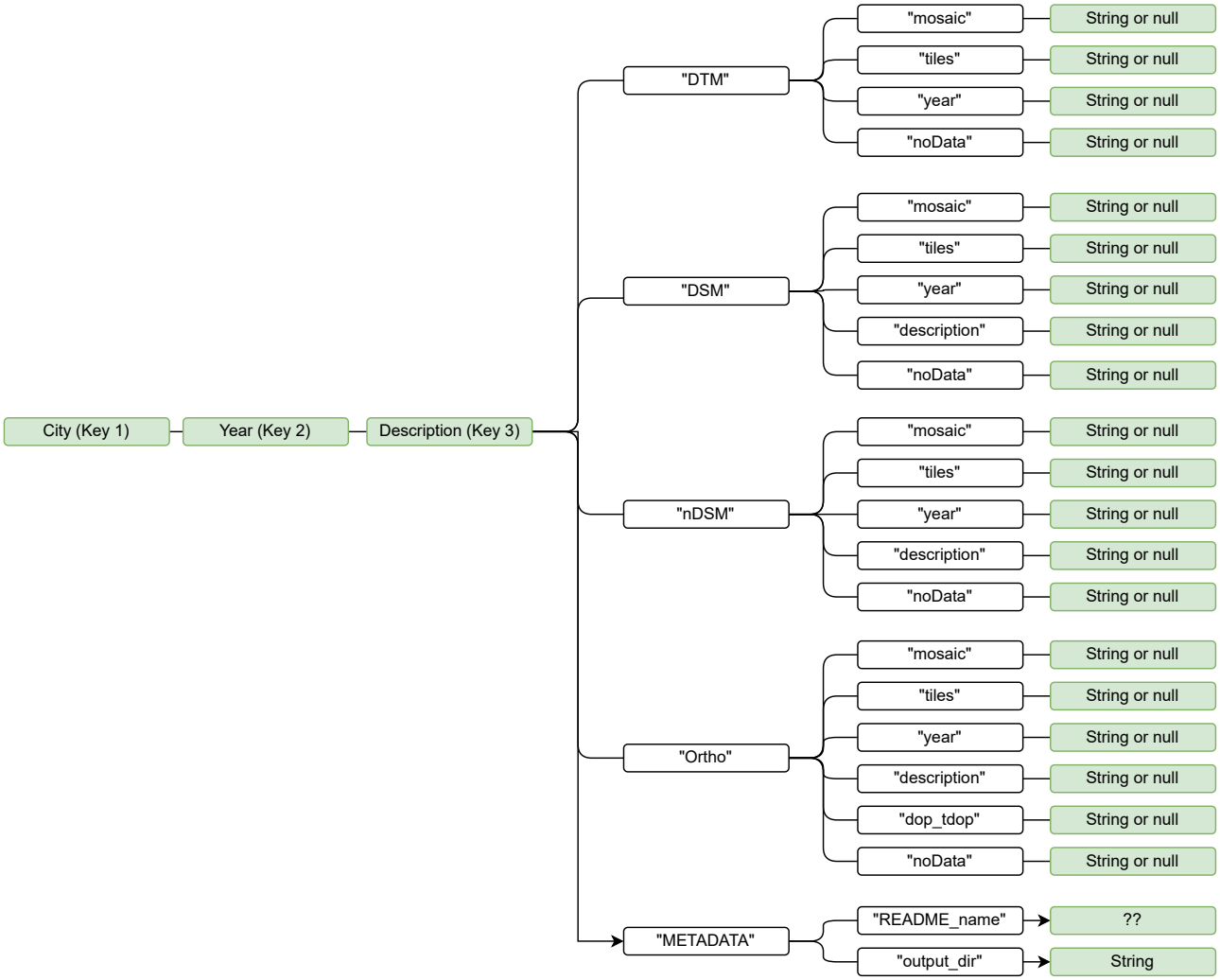



Figure 4: Model for the InputFilePointer JSON file

it is crucial to utilize a distinct combination of these three keys for each dataset. Failure to do so might result in the loss of a dataset, as there is no built-in mechanism to prevent such occurrences.

For each dataset, various values need to be specified, which provide information about the incoming rasters. In Figure 4, all the keys and values requiring configuration are enclosed within colored boxes.

Additionally, the following considerations should be taken into account:

- Only one of either ”mosaics” or ”tiles” should contain a path, with the other being assigned a null value.
- Either an nDSM or DSM+DTM path must be provided, while the other path(s) remains null.
- Mosaic paths should point to the mosaic file, whereas ”tiles” should point to the folder encompassing all the files, including subfolders.
- The ”noData” field is optional and defines the input nodata value for the ”tiles” (not for the mosaic). This parameter should only be defined if the value is not already specified in the file’s header, or if the file does not support a header (e.g., .xyz files).
- The keys within the 1-3 level hierarchy are primarily employed for naming the stack. On the other hand, attributes for DTM, DSM, nDSM, and ORTHO are used to label the mosaics. This can be particularly useful, for instance, if

the DTM originates from 2008, but the stack should be labeled as 2010 because the DSM and Ortho data pertain to 2010.

- The "dop_tdop" parameter indicates whether the ORTHO image is a true digital orthophoto or not.
- The "README" name establishes a name mentioned in a README file, which is included in the stack output repository for documentation purposes. It's recommended to use the name of the individual responsible for executing the pipeline. This way, they can be contacted in case of any issues that arise.

Please keep in mind that the JSON structure is nested. If multiple sets of the same city exist, they should be separate instances under the corresponding city key.

To verify consistency, an example JSON file is provided for comparison purposes.

### 5.4.2 CRSMetadatafile

This CSV file serves as a straightforward repository for storing EPSG codes corresponding to each dataset. The prefered location to store it is: root ¿¿ data ¿¿ CRSMetadata. It is structured based on the JSON InputFilePointer format and should utilize the same key identifiers as the JSON file. Unlike distinguishing between tiles and mosaic, this CSV doesn't differentiate between the two, as only one of them is expected to be provided. Therefore, the EPSG code pertains to the available option between tiles and mosaic. It's worth noting that, although the JSON requirement is focused on DSM+DTM or nDSM, it's acceptable to include an EPSG code for a non-existent dataset if it simplifies the process of creating this file. The csv looks like:

```
city,year,desc,DSM,DTM,nDSM,ORTHO
Augsburg,2022,unbelaub,25832,25832,,25832
Augsburg,2023,,25832,25832,,25832
```

### 5.4.3 Config.py

This file should only be modiefied by advanced users with programming knowledge.

#### condaenvironment

name of the conda/mamba environment

#### path_activate_conda

path to the conda/mamba activate.bat file i.e."C:/Users/QuadroRTX/miniconda3/condabin/activate.bat"

#### unused_cpus

Count of CPU cores to be reserved exclusively for tasks outside the pipeline. This allows for concurrent use of the machine during pipeline execution but might slightly reduce processing speed.

**chunksize**

Geowombat chunk size. No additional effort has been invested in determining an optimal chunk size for a machine. Feel free to experiment and find the most suitable configuration.

**stack_resolution**

Define the output stack resolution. This affects the stack and only the stack.

## 5.5 Temp

This module is mainly a folder with a certain folder structure created at the beginning of the pipeline to save all intermediate results. This mainly covers steps in the tile conversion. Folders are deleted after mosaic creation and at the absolute end to avoid storage issuses.

## 5.6 Output

A set folder structure is formed at a chosen location defined in the InputFilePointer JSON. If a folder for the same year exists, it automatically adds _2, _3 as suffixes to avoid storing two datasets in one folder. This is where the stack and mosaics are written to. The folder structure looks like

```
outputfolder
└── city
    └── year(_2,_3,...)
        ├── stack.tif
        ├── mosaic1.tif
        ├── mosaic2.tif
        ├── ...
        └── README.md
```

City and year folder are created automatically. Datasets belonging to the same city are stored in the same city folder. README.md is a template to document stuff in a structured way afterwards.

## 5.7 File naming convention

Files are automatically named based on metadata sourced from the JSON file. The combination of the three main keys is utilized for stack naming. Moreover, the 'tdop_dop' information is embedded within the filename. During pipeline execution, resolution is determined and applied for naming . Additionally, an determination is made for the number of bands in the ortho image. If there are three bands, the naming includes "RGB"; if 4, "RGBI" is used.

Mosaics are predominantly formed by leveraging attributes associated with DTM, DSM, nDSM, and ORTHO. Only the 'city' key is employed from the dataset keys. All these steps collectively contribute to the establishment of a standardized naming structure.

# 6 Runtime View

**Runtime Scenario 1**

The GUI is executed trough the exe.bat file. This requires a setup of the system. Espacially the installation of a conda environment and the definition within the config.py are crucial. Than the input data is created within the GUI. There is no need to open or run a python script manually

**Runtime Scenario 2**

The GUI.py is run from within a conda environment. Than the input data is created within the GUI.

**Runtime Scenario 3**

The Input files are created manually. Than the pipeline can be executed directly from within the script using a conda environment

**Runtime Scenario 4**

The pipeline is not used at all but the processing functions are used sololy. This requires loading them in a user created python script.

# 7 Results

Here could be some conclusions and recommendations for further development.