Internship Report

Optimizing the End-to-End Performance of Large-Scale Raster Processing in a Cloud Environment

GEO5016: Geomatics in Practice

Ming-Chieh Hu

Delft University of Technology

TU Delft

Internship Report

Optimizing the End-to-End Performance of Large-Scale Raster Processing in a Cloud Environment

by

Ming-Chieh Hu

Instructor:	J. Stoter
Internship Supervisor:	S. Briels and C. Cáceres
Geomatics Tutor:	M. Meijers
Project Duration:	July, 2025 - October, 2025
Faculty:	Faculty of Architecture & the Built Environment, Delft

Cover:	Aerial image from TrueOrtho overlapped with building footprints
Style:	TU Delft Report Style, with modifications by Ming-Chieh Hu

This report forms the written record of an internship carried out in the period *1 July 2025 - 31 October 2025*. The internship is an elective of the MSc Geomatics for the Built Environment, Delft University of Technology, awarded with 10 ECTS.

TU Delft

Abstract

This internship addresses significant performance bottlenecks in Readar's large-scale raster processing pipeline, which handles terabytes of high-resolution aerial imagery for the entire Netherlands. The research aims to optimize the data pipeline by improving both computational algorithms and I/O infrastructure. The work focused on two main tasks.

Task 1 optimized zonal statistics calculation for post-processing. A novel Hybrid method was proposed, which selects between a Vector-Raster (baseline) and a Raster-Raster (proposed) algorithm based on the local polygon "Occupancy Rate" of a tile. This Hybrid method consistently matched or outperformed the base methods, achieving speedups of up to 733% on dense data tiles. A key finding was that spatial tiling itself provided substantial performance gains by enabling sparse regions to be skipped.

Task 2 investigated I/O bottlenecks for deep learning data loading by benchmarking various raster formats (e.g., GeoTIFF, COG, Zarr) and cloud storage environments. Experiments demonstrated that the storage backend was the most critical factor: Azure Data Lake provided a $6 - 7\times$ read performance speedup over a mounted NAS. Tiled TIFF was the optimal format for random-access, confirming internal tiling is essential for this use case.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
API	Application Programming Interface
Blob	Binary Large Object
CLI	Command Line Interface
COG	Cloud Optimized GeoTIFF
CRS	Coordinate Reference System
DTM	Digital Terrain Model
GeoTIFF	Geographic Tagged Image File Format
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
NAS	Network-Attached Storage
PC	Personal Computer
PV	Photovoltaic
SRID	Spatial Reference Identifier
TIFF	Tag Image File Format
VM	Virtual Machine
VRT	Virtual Raster
VSI	Virtual System Interface

1

Introduction

1.1. Company

Readar[1] is a company of a startup scale and a tight organization structure. The focus of the company is to provide information from aerial imagery, using remote sensing and machine learning technologies.

The main products include:

- TrueOrtho: Orthorectified top view aerial photos
- Asbestos roof detection
- Green roof detection
- Photovoltaic panel detection
- PV-potential detection
- Change detection

Many Dutch municipalities are using Readar's data, to monitor for instance, asbestos roofs, solar panels and green roofs. Readar's data is also used by several industries for a large variety of applications. For example, insurance companies use the data for calculating fees, while installation companies leverage it for lead generation and providing quotes. Furthermore, banks rely on the data for property valuations, energy companies apply it for marketing purposes, and utility companies utilize it for monitoring their operations.

1.2. Department

The core technical work is handled by an integrated team. This group is fully responsible for the entire data pipeline, from managing raw aerial imagery to delivering the final data products, including ongoing maintenance and development. This end-to-end workflow is functionally divided into the following key areas:

- Pre-processing: Preparing raw images for the pipeline, which includes generating attributes and reformatting raster files for optimized access.
- Post-processing: Finalizing the prediction from deep learning models, e.g., clipping images and generate statistical result from raster to csv tables.
- Quality Control (QC): Working with pre-processed data to perform quality checks, detect and clear corrupted data, and provide feedback to the external aerial photo provider.
- Deep Learning Model Development: Developing and improving models based on client demand. This ensures continuous innovation on the core product.

I worked closely with the data processing and quality control components, aiming to improve the runtime performance, as will be detailed in the following section.

[1] readar.com

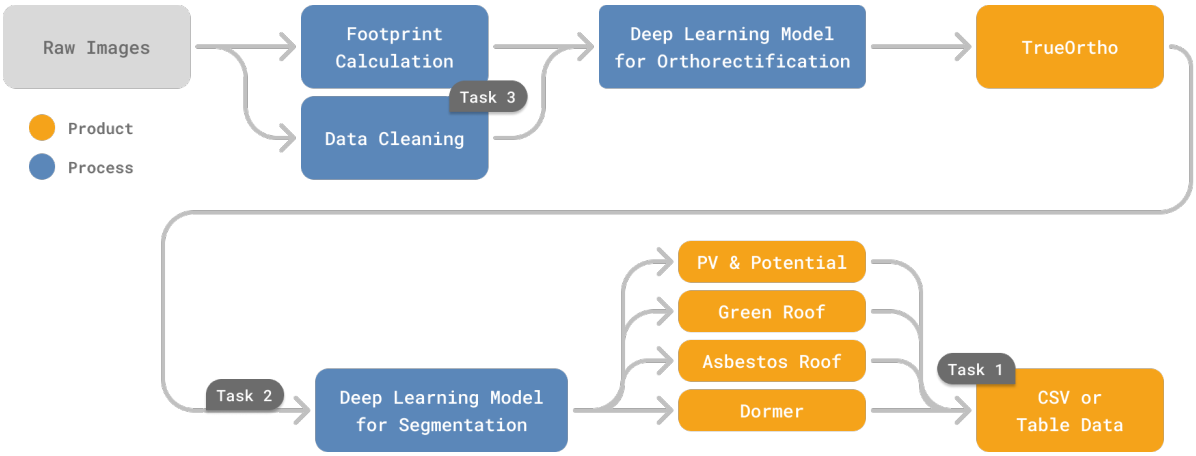


Figure 1.1: Data processing pipeline overview

1.3. Internship Topic

Problem Statement Readar processes data for the entire Netherlands, which involves handling terabytes of raster images with a resolution as high as 5cm. The huge volume and high resolution of this raster data create multiple throughput bottlenecks in the data processing pipeline, as shown in Figure 1.1. These bottlenecks can obstruct the efficiency of the maintenance, operational tasks and the development of new products. The problems stem from two core issues:

- I/O Inefficiency: The existing data infrastructure, particularly the storage format, is not optimized for random-access tasks. For example, it is slow to clip raster data using building footprints to create training datasets, making it unfeasible to process the data for the entire country in a timely manner.
- Computational Inefficiency: Even with parallelism, the existing algorithms for analyzing such high-resolution data can still be a major bottleneck, limiting the speed at which final data products is generated.

The research question is hence **"To what extent can the performance of a geospatial data processing pipeline be improved by optimizing both infrastructure (storage formats and I/O) and algorithms?"**

Goals and Tasks My internship focused on benchmarking/resolving these performance bottlenecks. The three identified bottlenecks were marked with Task 1, 2 and 3, as shown in Figure 1.1.

Task 1: Optimizing Zonal Statistics This task tackled a bottleneck in the post-processing steps, which converts the segmented raster (raster with integer values representing categories) into statistical results grouped by buildings. The goal was to find a practical solution for the PV statistics calculation that best balanced time efficiency and memory consumption. The optimized solution can also be applied to other data products.

Task 2: Investigating Raster I/O Bottlenecks To create the training, testing, and validation datasets for deep learning models, images must be extracted (or "cut out") using building footprints before being parsed into a data loader. This data creation process itself constitutes a major bottleneck within the training phase of deep learning models. The task involved a systematic evaluation of modern raster file formats (e.g., GeoTIFF, COG, Zarr) to benchmark their performance for random-access tasks in Readar's cloud environment. The objective was to identify the optimal storage strategy and file format to reduce data access times.

Task 3: Application for Feature Creation and QC The final task was to synthesize the findings from the previous tasks into an accelerated application for the data preparation process before the orthorectification step. This involved developing a GUI and CLI tool for data creation, inspection, and quality control (specifically, footprint calculation and misalignment detection). This tool utilizes the optimized file formats and parallel processing methods to replace the existing, obsolete script.

For better readability, the core experimental work, covering Task 1 and Task 2, is presented sequentially in Chapter 2 and 3. Task 3, which provides context on a related company application without an experimental evaluation, is detailed in Appendix A.

2

Task 1: Optimizing Zonal Statistics

Task 1 involves calculating and generating statistics on the coverage of PV panel installations on building rooftops, primarily by measuring the amount of pixel. This is a zonal statistics problem, which aggregates all the values from the raster layer that overlap with a set of polygons. Think of it like a cookie cutter: classified rasters are "cut" using numerous building polygons to isolate and analyze only the relevant areas.

Traditional methods to process the zonal statistics problem focused on either vectorizing the raster [10] or rasterizing the vector [3] data. ScanLine [2] method is a newer and faster approach, which processes the raster and vector data without a conversion process.

In this part, two new methods are proposed, the Raster-Raster and the Hybrid method; the first rasterize the polygons in batches and does a array multiplication at once, while the Hybrid method takes the advantage of the baseline and Raster-Raster method based on the building occupancy rate per tile. The proposed methods and the baseline solution are based on GDAL and Rasterio. Rasterio's (and hence GDAL's) internal C++ implementation uses a scanline approach to turn the vector geometries into the raster efficiently. Therefore, the result is still a "rasterization", but the scanline algorithm is actually the core of it.

There also exists other possibilities to further speed up the process, e.g., distributed systems for big data [6, 7], GPU acceleration, but they will not be discussed in this task as this task focuses on the single machine, non-GPU-based solution.

2.1. Algorithms

2.1.1. Vector-Raster Method (Baseline)

As shown in Algorithm 1, the method performs a series of operations in parallel: the raster file is opened and clipped to the extent of the polygon geometry, after which the resulting masked image is converted to a binary map where pixels equal to 3 become 1 and all others become 0. The pixels with value 1, representing photovoltaic areas, are counted, and the building's identifier, PV pixel count, and total pixel count are recorded. The results from all buildings are then compiled into a DataFrame for further analysis or storage.

Algorithm 1 Baseline PV Detection

Require: Raster $\mathcal{R}$, Set of polygons P, Target value v
Ensure: DataFrame $\mathcal{D}$ with pixel statistics per polygon

1: **for all** $p \in P$ **do** ▷ Can be processed in parallel
2: $I_{crop} \leftarrow \textsc{Crop}(\mathcal{R}, p)$ ▷ Clip raster to polygon extent
3: $count \leftarrow \sum[I_{crop} = v]$ ▷ Count pixels matching target value
4: $total \leftarrow \textsc{CountPixels}(I_{crop})$
5: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(id_p, count, total)\}$
6: **end for**
7: **return** $\mathcal{D}$

2.1.2. Raster-Raster Method (Proposed)

The baseline method is simple and performs well with a small number of building polygons. However, beyond a certain threshold, performance degrades significantly because each polygon requires a separate raster file I/O operation.

When dealing with a very large number of polygons, it is generally more efficient to convert the vector data into raster format. This approach benefits from a technique known in array programming as vectorization, which refers to performing operations on entire arrays without explicit loops. By replacing iterative processes with vectorized operations, we can take advantage of NumPy's underlying C implementation for faster computation.

The task is transformed from a vector–raster intersection to a raster-to–raster, array multiplication. It utilizes `rasterio.mask`, creates an integer map, and reads the raster file in tiles, performing raster multiplication tile by tile. The general framework is illustrated in Algorithm 2.

Algorithm 2 Rasterized PV Detection

Require: Raster $\mathcal{R}$, Set of polygons P, Target value v
Ensure: DataFrame $\mathcal{D}$ with pixel statistics per polygon

$L \leftarrow \textsc{InitZeroRaster}(\text{Bounds}(P))$ ▷ Initialize label map
for all $p \in P$ **do**
  $L \leftarrow \textsc{BurnGeometry}(L, p, id_p)$ ▷ Rasterize polygon ID into map
  $\mathcal{D}[id_p].total \leftarrow \textsc{CountNonZero}(p)$
end for
$Data \leftarrow \textsc{Read}(\mathcal{R})$
$Matches \leftarrow (Data = v)$ ▷ Boolean mask of target pixels
$counts \leftarrow \textsc{Bincount}(L, \text{weights} = Matches)$ ▷ Vectorized counting
for all $id \in \text{Unique}(L)$ **do**
  $\mathcal{D}[id].count \leftarrow counts[id]$
end for
return $\mathcal{D}$

2.1.3. Hybrid Method (Proposed, Best-performing)

As shown in Section 2.2, building distribution across the study area is highly heterogeneous, with covered areas varying significantly within individual tiles. A single tile may contain regions that are nearly empty—ideally suited for the Vector-Raster method—alongside densely built areas where the Raster-Raster method performs better. Hence, a hybrid method is proposed to take advantage of the both, Figure 2.1 and Algorithm 3 depicted the proposed workflow.

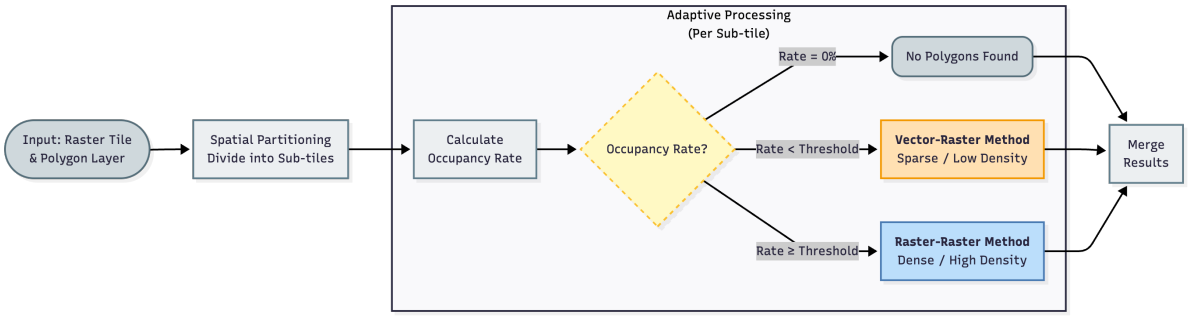


Figure 2.1: Workflow of the proposed Hybrid method

Adaptive Method Selection A straightforward approach would involve establishing a polygon count threshold to determine method selection. However, given the substantial variation in tile sizes shown in Figure 2.3, simple polygon count thresholds prove unreliable. A more robust solution employs *occupancy rate* as the decision metric:

$$OccupancyRate = \frac{\text{Area(polygons)}}{\text{Area(raster)}} \times 100\% \tag{2.1}$$

For each sub-tile, the processing strategy follows:

$$\text{Do} \begin{cases} \textbf{Nothing}, & \text{if } \textit{no building polygon found} \\ \textbf{Vector-Raster method}, & \text{if } OccupancyRate < threshold\ \% \\ \textbf{Raster-Raster method}, & \text{if } OccupancyRate \geq threshold\ \% \end{cases} \tag{2.2}$$

Spatial Partitioning (Tiling) To maximize the effectiveness of this adaptive approach, we implement hierarchical spatial partitioning by subdividing tiles into sub-tiles. This finer granularity also serves a critical purpose, as it creates regions that approach the extreme cases where each base method excels—either sparsely populated areas favoring Vector-Raster processing or densely built zones optimal for Raster-Raster operations.

This partitioning strategy enables the algorithm to select the more efficient processing method based on local polygon density rather than global tile properties, resulting in significant performance improvements across diverse urban patterns.

Algorithm 3 Hybrid Tiled Processing

Require: Raster $\mathcal{R}$, Set of polygons P, Target v, Threshold τ
Ensure: Accumulator $\mathcal{D}$ (initialized to 0 for all $p \in P$)

Partition $\mathcal{R}$ into a grid of tiles $T = \{t_1, t_2, \dots, t_n\}$
for all $t \in T$ **do**
  Define geographic bounds B_t of tile t
  $P_{local} \leftarrow \{p \in P \mid p \cap B_t \neq \emptyset\}$ ▷ Find intersecting polygons
  if $P_{local} = \emptyset$ **then**
    continue
  end if
  $\rho \leftarrow \textsc{OccupancyRate}(P_{local}, B_t)$
  if $\rho < \tau$ **then** ▷ Low density: Use Baseline
    $R_{local} \leftarrow \textsc{VectorRaster}(P_{local}, \mathcal{R}, v)$
  else ▷ High density: Use Raster-Raster
    $W_t \leftarrow \textsc{GetWindow}(\mathcal{R}, t)$
    $R_{local} \leftarrow \textsc{RasterRaster}(P_{local}, W_t, v)$
  end if
  $\mathcal{D} \leftarrow \mathcal{D} + R_{local}$ ▷ Accumulate partial results
end for
return $\mathcal{D}$

2.2. Data

The dataset used in this task is designed to measure the area of installed PV panels, therefore all files are with only one band. Four sample tiles were selected to test various scenarios. Their overall statistics are presented in Table 2.1, and visualizations are provided in Figures 2.2 and 2.3. Accordingly, the files follow the naming format `pv_xx_yy.tif`. Throughout this task, this format will be used, or simply abbreviated as tile `xx_yy`.

Each tile typically covers an area of around one hundred million square meters, with dimensions exceeding $100,000 \times 100,000$ pixels. Additionally, the distribution of buildings can differ from tile to tile; this variation is addressed using the $OccupancyRate$ defined in Eq. 2.1.

Tile	Buildings No.	Building Area (m^2)	Tile Area (m^2)	Occupancy Rate	File Size
`pv_18_42.tif`	102,584	9,422,577.69	99,656,663.04	9.46%	82 MB
`pv_18_43.tif`	54,215	5,103,095.15	102,770,933.76	4.97%	71 MB
`pv_20_50.tif`	60,522	6,309,149.57	99,656,663.04	6.33%	69 MB
`pv_24_52.tif`	13,008	1,879,524.32	99,656,663.04	1.89%	59 MB

Table 2.1: Building coverage statistics for selected tiles

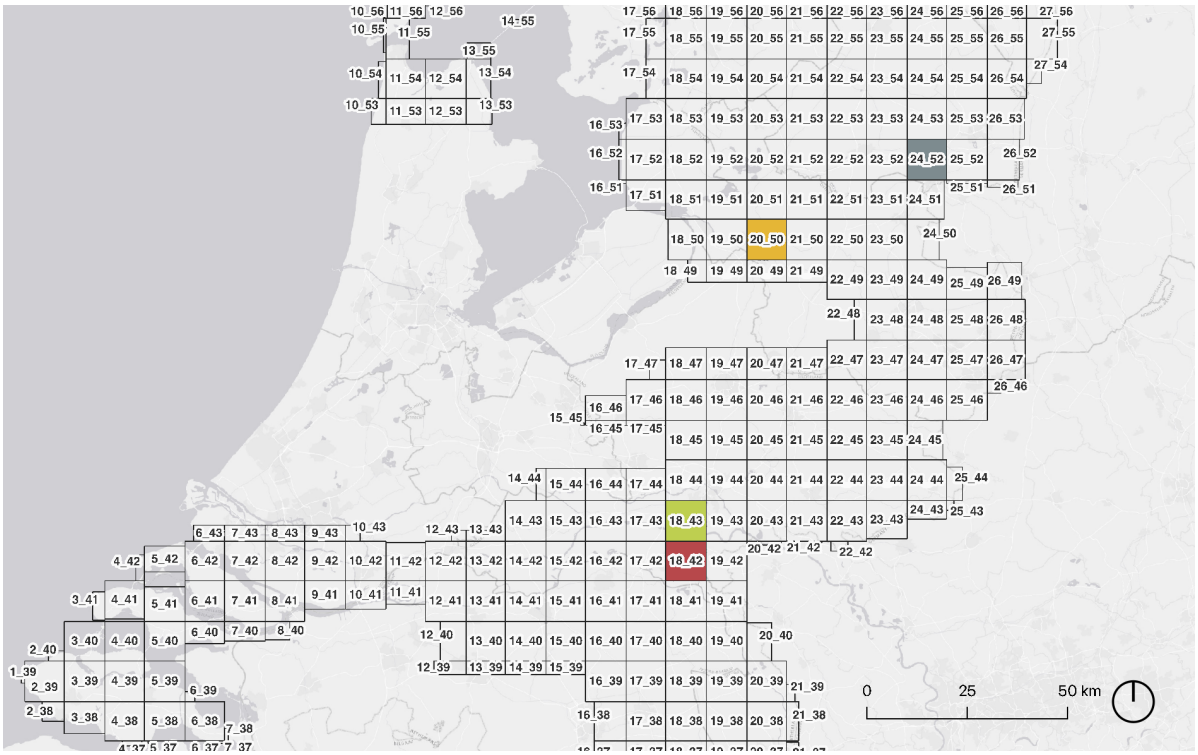
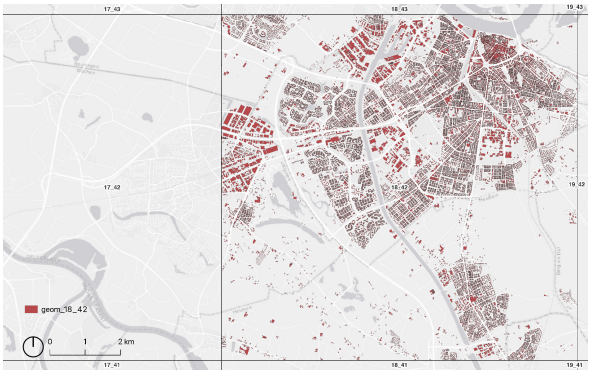
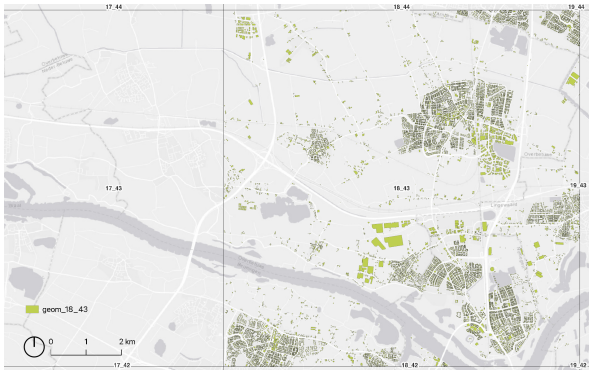


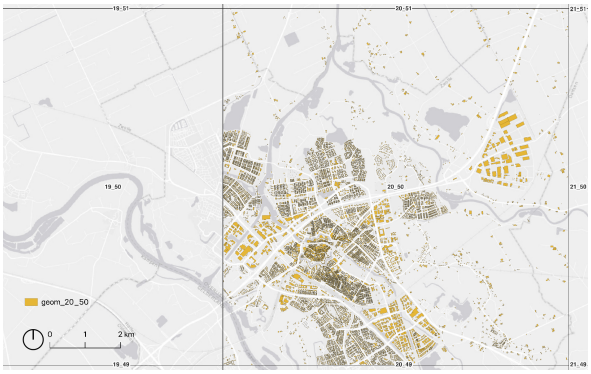
Figure 2.2: Task 1 dataset: tile division overview



(a) Building polygons in tile 18_42



(b) Building polygons in tile 18_43



(c) Building polygons in tile 20_50

(d) Building polygons in tile 24_52

Figure 2.3: Task 1 dataset: building footprints in the extent of the 4 tiles

2.3. Experiments

While the Vector–Raster Method (baseline) can operate effectively without spatial partitioning, the Raster–Raster Method requires substantial memory resources that exceed available system capacity when processing entire tiles simultaneously. To enable fair comparison, all methods are evaluated using the same tiling approach, which also allows the Hybrid Method to achieve performance gains through adaptive method selection based on polygon occupancy rate within sub-tiles.

To isolate the algorithmic contributions from tiling effects, the evaluation consists of two components:

1. **Performance evaluation across tiling schemes**: Evaluate all three methods (Vector-Raster, Raster-Raster, and Hybrid) across a range of tiling schemes from 1×1 to 32×32 sub-tiles. This analysis determines each method's performance characteristics across different spatial granularity and identifies optimal tiling configurations for each method.
2. **Parameter optimization**: Evaluate and determine the optimal $OccupancyRate$ threshold for the Hybrid Method.

This experimental design ensures that performance comparisons reflect genuine algorithmic differences rather than memory limitations. The results and analysis are presented in Section 2.4.

2.4. Results

2.4.1. Comprehensive Evaluation Across Different Tiling Schemes

Currently, we have several variables to consider: the $OccupancyRate$ threshold, the tiling scheme, and the choice of method. To enable fair comparisons, we made a reasonable assumption: the $OccupancyRate$ threshold should lie between 1.89% (from tile `24_52`) and 9.46% (from tile `18_42`), based on their observed overall occupancy rates. After examining Table 2.1, we selected a threshold of 5%, which will be further evaluated in Section 2.4.2.

The comprehensive evaluation tests all three methods (Vector-Raster, Raster-Raster, and Hybrid) across a range of tiling schemes, from 1×1 to 32×32 sub-tiles. The corresponding approximate tile sizes are 120000×120000, 60000×60000, 30000×30000, 15000×15000, 7500×7500, and 3800×3800.

As shown in Table 2.2, the proposed Hybrid method performs significantly better than the other methods on tile `18_42`, while delivering comparable or slightly improved results on tiles `18_43`, `20_50`, and `24_52`. Here, "OOM" indicates an out-of-memory error. Additional analysis plots are provided in Appendix C.

Interestingly, chunking large rasters into smaller tiles yields substantial runtime improvements—even greater than the gains from combining the two methods into a hybrid approach. For instance, the Vector-Raster baseline method takes 985 seconds without tiling, but only 216 seconds with 8×8 tiling. This trend holds across most files. The improvement is due not only to the higher chance of skipping sparse tiles at finer granularity but also to the non-linear reduction in the time complexity of parallel task allocation. For the Raster-Raster method, the runtime also improves since the reduction in file I/O time outweighs the added loop overhead. However, there is clearly a point of diminishing returns... pushing to 32×32 subdivision often degrades performance again, likely because the overhead of managing too many small tiles starts to outweigh the computational benefits.

Tile `24_52` consistently shows the fastest processing (around 2-4ms per million pixels for most configurations), while tile `18_42` requires significantly more computation time (up to 63ms per million pixels without tiling). This dramatic difference aligns with their occupancy rates and suggests that sparse tiles with lower occupancy are inherently easier to process, regardless of the method used.

As expected, the Vector-Raster method performs better when the $OccupancyRate$ is low, whereas the Raster-Raster method is more efficient at higher occupancy levels. The Hybrid method consistently outperforms both by adapting to the characteristics of the data and leveraging the strengths of each approach. For our use case file size, it is found that the 16×16 tile division is the best performing one, and that roughly equals to 7500×7500 in tiling pixel size. The hybrid approach never falls below the performance of the better base method, which validates the assumption that method selection based on local occupancy can indeed provide improvements across diverse data characteristics.

Tile	Method	Runtime	1×1	2×2	4×4	8×8	16×16	32×32
18_42	Vector-Raster	Total (s)	985.505	398.752	245.695	216.634	253.420	323.891
		Per Million Pixels (ms)	63.290	25.608	15.779	13.912	16.275	20.800
	Raster-Raster	Total (s)	OOM	337.848	296.932	227.194	186.532	188.549
		Per Million Pixels (ms)	OOM	21.697	19.069	14.590	11.979	12.109
	Hybrid	Total (s)	OOM	238.201	217.938	148.166	**134.384**	154.702
		Per Million Pixels (ms)	OOM	15.297	13.996	9.515	**8.630**	9.935
		Total Pixels						15,571,353,600
18_43	Vector-Raster	Total (s)	281.743	147.008	115.622	115.935	134.343	180.174
		Per Million Pixels (ms)	17.545	9.155	7.200	7.220	8.366	11.220
	Raster-Raster	Total (s)	OOM	298.452	265.435	190.590	171.499	134.049
		Per Million Pixels (ms)	OOM	18.586	16.530	11.869	10.680	8.348
	Hybrid	Total (s)	281.743	236.581	167.613	103.805	**95.701**	101.246
		Per Million Pixels (ms)	17.545	14.733	10.438	6.464	**5.960**	6.305
		Total Pixels						16,057,958,400
20_50	Vector-Raster	Total (s)	473.175	255.222	156.194	137.599	182.913	210.498
		Per Million Pixels (ms)	30.388	16.390	10.031	8.837	11.747	13.518
	Raster-Raster	Total (s)	OOM	321.916	282.401	213.208	161.890	130.942
		Per Million Pixels (ms)	OOM	20.674	18.136	13.692	10.397	8.409
	Hybrid	Total (s)	OOM	140.365	165.669	**101.345**	**101.555**	112.995
		Per Million Pixels (ms)	OOM	9.014	10.639	**6.508**	**6.522**	7.257
		Total Pixels						15,571,353,600
24_52	Vector-Raster	Total (s)	35.026	34.996	33.421	37.349	50.348	66.506
		Per Million Pixels (ms)	2.249	2.247	2.146	2.399	3.233	4.271
	Raster-Raster	Total (s)	OOM	276.111	238.266	186.414	125.425	82.681
		Per Million Pixels (ms)	OOM	17.732	15.302	11.972	8.055	5.310
	Hybrid	Total (s)	35.026	**31.998**	38.161	40.804	44.051	47.890
		Per Million Pixels (ms)	2.249	**2.055**	2.451	2.620	2.829	3.076
		Total Pixels						15,571,353,600

Table 2.2: Comprehensive evaluation across different tiling schemes

2.4.2. Parameter Optimization Across Different $OccupancyRate$ Thresholds

As the effect of tiling scheme is displayed in Section 2.4.1, the best-performing parameter to divide tiles is fixed to be 16×16. In this part, we further evaluate the Hybrid method using candidate $OccupancyRate$ thresholds of 0.02, 0.03, 0.04, 0.05, 0.06, 0.07 and 0.08.

From the Tables 2.3, 2.4, 2.5 and 2.6, one can see that the best threshold value varies depending on the spatial distribution of buildings within each tile. Notably, only tile 18_42 demonstrates improved performance when utilizing a higher proportion of the Raster-Raster method, while the other tiles perform better with greater reliance on the Baseline method. tile 18_42 achieves minimum runtime at 0.06 threshold, while tiles 18_43 and 20_50 perform best at 0.05 and 0.08 respectively, and tile 24_52 shows optimal performance at 0.07.

This variation reflects the adaptive nature of the hybrid approach, where tiles with different building densities and spatial patterns benefit from different decision boundaries between the Baseline and Raster-Raster Methods. However, the performance differences between threshold values are relatively small, with runtime variations typically within 10-15% of the optimal value. Figures C.5, C.6, C.7 and C.8 showcase the detailed plots of the proportion of each case.

Metric \ Threshold	0.02	0.03	0.04	0.05	0.06	0.07	0.08
Tiles Processed with Baseline (%)	25.8	34.0	39.1	41.0	43.4	44.5	46.9
Tiles Processed with Solution 1 (%)	66.8	58.6	53.5	51.6	49.2	48.0	45.7
Tiles Skipped (%)	7.4	7.4	7.4	7.4	7.4	7.4	7.4
Runtime: Baseline (s)	10.900	18.525	22.892	25.616	28.721	30.442	35.268
Runtime: Solution 1 (s)	125.441	123.893	109.959	105.627	100.357	102.071	94.030
Runtime: Total (s)	147.787	154.332	144.599	142.793	**140.426**	144.001	**140.517**
Runtime Per Million Pixels (ms)	9.491	9.911	9.286	9.170	**9.018**	9.248	**9.024**
Total Pixels							15,571,353,600

Table 2.3: $OccupancyRate$ threshold analysis of tile `18_42`

Metric \ Threshold	0.02	0.03	0.04	0.05	0.06	0.07	0.08
Tiles Processed with Baseline (%)	42.2	50.0	53.1	56.6	61.3	64.1	65.6
Tiles Processed with Solution 1 (%)	47.7	39.8	36.7	33.2	28.5	25.8	24.2
Tiles Skipped (%)	10.2	10.2	10.2	10.2	10.2	10.2	10.2
Runtime: Baseline (s)	16.556	22.019	24.427	28.531	36.547	39.957	45.169
Runtime: Solution 1 (s)	92.207	79.998	71.493	66.652	62.438	55.585	53.913
Runtime: Total (s)	117.400	110.513	104.184	**103.451**	107.556	**104.080**	107.558
Runtime Per Million Pixels (ms)	7.311	6.882	6.488	**6.442**	6.698	**6.482**	6.698
Total Pixels							16,057,958,400

Table 2.4: $OccupancyRate$ threshold analysis of tile `18_43`

Metric \ Threshold	0.02	0.03	0.04	0.05	0.06	0.07	0.08
Tiles Processed with Baseline (%)	44.5	53.5	57.4	58.2	59.8	61.7	62.5
Tiles Processed with Solution 1 (%)	48.0	39.1	35.2	34.4	32.8	30.9	30.1
Tiles Skipped (%)	7.4	7.4	7.4	7.4	7.4	7.4	7.4
Runtime: Baseline (s)	14.721	21.941	25.225	26.157	28.112	28.966	30.264
Runtime: Solution 1 (s)	90.758	75.132	68.639	68.231	65.156	59.117	57.496
Runtime: Total (s)	114.718	106.420	103.111	103.446	102.286	**96.924**	**96.658**
Runtime Per Million Pixels (ms)	7.367	6.834	6.622	6.643	6.569	**6.225**	**6.207**
Total Pixels							15,571,353,600

Table 2.5: $OccupancyRate$ threshold analysis of tile `20_50`

Metric \ Threshold	0.02	0.03	0.04	0.05	0.06	0.07	0.08
Tiles Processed with Baseline (%)	56.6	64.1	66.8	68.4	71.5	71.5	72.7
Tiles Processed with Solution 1 (%)	21.5	14.1	11.3	9.8	6.6	6.6	5.5
Tiles Skipped (%)	21.9	21.9	21.9	21.9	21.9	21.9	21.9
Runtime: Baseline (s)	15.640	20.585	22.558	23.794	27.664	27.783	30.250
Runtime: Solution 1 (s)	34.660	23.469	19.253	16.795	11.896	11.947	9.865
Runtime: Total (s)	53.563	47.342	45.165	43.913	**43.234**	**42.939**	43.594
Runtime Per Million Pixels (ms)	3.440	3.040	2.901	2.820	**2.777**	**2.758**	2.800
Total Pixels							15,571,353,600

Table 2.6: $OccupancyRate$ threshold analysis of tile `24_52`

3

Task 2: Investigating Raster I/O Bottlenecks

Deep learning workflows in remote sensing, often implemented using the PyTorch framework and specialized libraries like TorchGeo [8], are frequently constrained by data loader I/O bottlenecks. While TorchGeo provides a valuable toolkit for model training, it does not explicitly address the performance of various storage environments or cloud-based data access patterns. Furthermore, prior work by Zaytar et al. [9], which focused on improving GeoTIFF loading throughput from cloud object storage to GPU, emphasizing tile-aligned reads and cache strategies rather than optimizing for random access.

Task 2 addresses the critical I/O bottleneck by comparing raster file formats using controlled experiments across representative production environments. Specifically, the evaluation assesses the performance of GeoTIFF [4] with various access and tiling configurations, alongside the cloud-optimized Zarr format [1]. Nine distinct combinations of file format and tiling schemes were selected, with each variant tested under different compression and predictor parameter combinations to examine both storage efficiency and read performance. Finally, all configurations are benchmarked across two distinct testing environments to assess how infrastructure characteristics affect runtime performance. An overview of these experimental parameters is displayed in Figure 3.1.

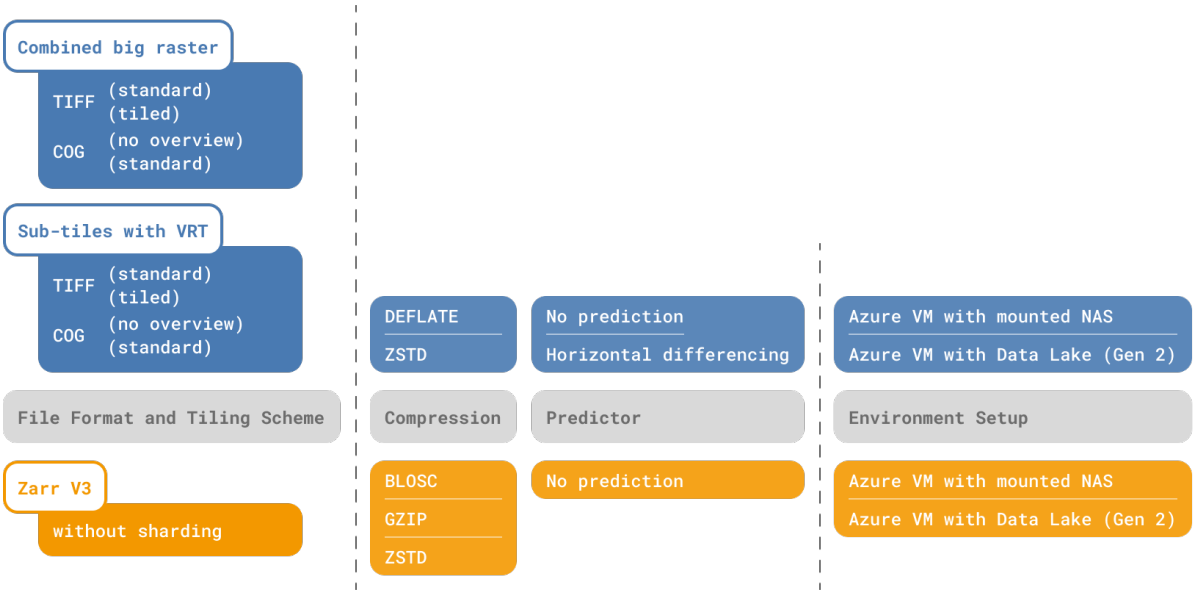


Figure 3.1: Overview of file format variants and test environment configurations (top: TIFF-based; Bottom: Zarr)

3.1. File Format Variants and Configurations

3.1.1. File Format and Tiling Scheme

Nine primary format configurations were evaluated to assess their suitability for the benchmarking task:

1. **Combined TIFF**: Standard TIFF with scanline organization, storing all data in a single monolithic file.
2. **Combined Tiled TIFF**: Single TIFF file with internal 256×256 pixel tiling for optimized spatial access patterns.
3. **Combined COG TIFF without overview**: Cloud-optimized TIFF implementation featuring 256×256 pixel internal tiling without pyramid overviews.
4. **Combined COG TIFF**: Cloud-optimized TIFF implementation featuring 256×256 pixel internal tiling with pyramid overviews for multi-resolution access.
5. **VRT + TIFF tiles**: Virtual raster (VRT) providing logical aggregation of multiple standard TIFF tile files.
6. **VRT + Tiled TIFF tiles**: Virtual raster referencing multiple tiled TIFF files, each with 256×256 pixel blocks.
7. **VRT + COG TIFF tiles without overview**: Virtual raster referencing multiple COG TIFF tile files without pyramid overviews.
8. **VRT + COG TIFF tiles**: Virtual raster referencing multiple COG TIFF tile files.
9. **Zarr V3**: N-dimensional arrays tiled externally according to the Zarr standard.

Note that the software implementation of this task in Zarr differs from that for TIFF-based files and can therefore impact the performance. The software implementation and issues encountered for benchmarking Zarr is detailed in Section D.2.

3.1.2. Parameters

The compression and predictor parameters are explained in Table 3.1. For Zarr files, it is possible to utilize horizontal differencing predictors, but this functionality is not from the official library, we therefore decide not to use this function.

Format	Parameter Type	Setting / Algorithm	Description
TIFF	Compression	DEFLATE	Standard ZIP-based compression algorithm
		ZSTD	Zstandard compression (improved ratio/speed)
	Predictor	1	No prediction applied (raw pixel values)
		2	Horizontal differencing (current pixel - previous)
Zarr	Compression	GZIP	DEFLATE-based with headers and check sum
		ZSTD	Zstandard compression (improved ratio/speed)
	Predictor	-	Not evaluated (not in official library)

Table 3.1: Comparison of Evaluated Parameters for TIFF and Zarr Formats

3.1.3. Environment Setup

The two testing environments represent different data access paradigms in a production environment:

1. **Azure VM[1] with Mounted NAS (Network Attached Storage)**: Network-attached storage configuration where data resides on a mounted network drive. This setup is not optimized for frequent file I/O operations, but is a typical scenario for companies managing big data.
2. **Azure VM and Data Lake[2]**: Cloud-native storage optimization leveraging HTTP range requests to fetch only required data portions from Azure Data Lake Storage. This approach minimizes transfer times by requesting precise byte ranges rather than entire files, making it specifically designed for efficient remote access to large geo-spatial datasets over internet connections.

3.2. Data

The selected dataset is a tile from the product TrueOrtho, with tile id `18_42`, which is part of a collection covering the Netherlands. This tile itself has been subdivided into 400 sub-tiles arranged in a 20 × 20 grid, resulting in a folder containing 400 individual TIFF files. The complete tile covers 125,000 × 125,000 pixels,

[1] Azure Virtual Machine: https://azure.microsoft.com/en-us/products/virtual-machines
[2] Azure Data Lake: https://azure.microsoft.com/en-us/solutions/data-lake

representing a spatial extent of 10,000m × 10,000m with a pixel size of 0.08m. The imagery is stored in ordinary TIFF format with DEFLATE compression and no predictor (PREDICTOR=1), and each sub-tile contains 4 bands configured as R, G, B, A. Here, "sub-tile" means the data is explicitly tiled into separate files. The specification is also shown in Table 3.2

Attribute	Value	Description
Format	TIFF	Ordinary TIFF format with no internal tiling
Compression	DEFLATE	Lossless compression method
Predictor	1 (None)	Default value, no predictive encoding
Bands	4 (R, G, B, A)	Red, Green, Blue, Alpha channels
Total Dimension	125,000 × 125,000 pixels	Complete tile dimensions before subdivision
Spatial Extent	10,000m × 10,000m	Ground coverage area
Pixel Size	0.08m	Ground sampling distance
Sub-tile Grid	20 × 20	Grid arrangement of sub-tiles (400 Individual `.tif` files)
Sub-tile Size	6,250 × 6,250 pixels	Each sub-tile dimensions
Sub-tile Coverage	500m × 500m	Ground area per sub-tile

Table 3.2: Reference data specifications - Tile 18_42

3.3. Experiments

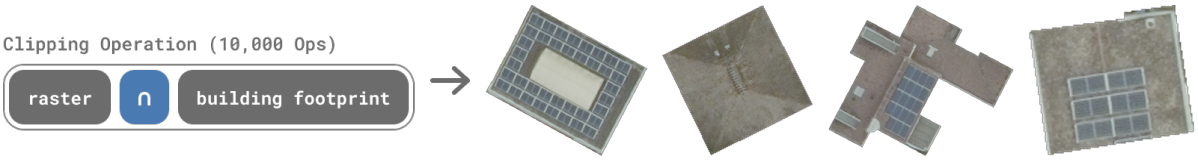


Figure 3.2: Benchmarking task explained in diagram

3.3.1. Benchmarking Task and Performance Metrics

To evaluate the performance impact of different raster formats on high-resolution aerial imagery processing, a controlled experiment was conducted to compare file I/O performance across multiple format configurations. The benchmark task was designed to closely reflect the company's use case: clipping and storing images using raster data and building footprint geometries within an area of interest (see Fig. 3.2).

Since multiple raster formats were assessed, format conversion time was also included in the evaluation. Although such conversions can be parallelized (e.g., across multiple virtual machines), measuring the conversion time provides insight into data creation costs and potential optimizations.

Accordingly, three primary metrics were measured to assess format performance characteristics:

1. **Operational Throughput**: Time to complete 10,000 data access operations, simulating typical raster clipping workflows used in the production pipeline. This is the primary metric of interest.
2. **Format Conversion Time**: Time required to convert source data into each target format configuration, representing a secondary metric for evaluating file formats.
3. **Storage Efficiency**: Final file size after compression, indicating storage and data transfer cost.

3.4. Results

In this section the experiment results for Task 2 are presented. Results ran on Azure VM with Mounted NAS, Azure VM with Azure Data Lake are displayed in Table 3.3 and 3.5 below.

3.4.1. TIFF-based Files

Comparison Across Different File Formats

Combined tiled TIFF files and Combined COG TIFF (without overviews) are the fastest to finish the benchmarking task across every test environment setup. They are also some of the faster file formats to finish format conversion. However, even with the raster pyramid overview turned off, the ordinary Combined tiled TIFF

is still the faster in both the conversion and the benchmarking task runtime. This shows that, other than the raster pyramid, there might be some minor difference in COG files, and it is unnecessary in this use case.

COG files are in general the slowest to create, taking 15-40× longer than the fastest option due to resampling. They are also the largest in terms of file size when using the same compression and predictor parameters, as the files contain additional data. In general, internally tiled files consistently outperform non-tiled formats for the benchmarking operations. But the raster pyramid in COG files is confirmed to be unnecessary in this use case.

Accessing files through VRT to actual sub-tiles is generally slower than accessing one combined file. The exception is accessing the Combined TIFF file, which performs poorly because that file format lacks internal tiling. This penalty is particularly severe with NAS due to internet latency, but less pronounced with Azure Data Lake.

Test Environments Specific Behavior

Across different test environments, all TIFF-based file formats has delivered similar behavior relatively, in both the conversion and the benchmarking task runtime. In general, accessing files from the local file system is faster than accessing from mounted NAS or Azure Data Lake, which is expected behavior.

Accessing one Combined TIFF versus accessing through VRT with 400 explicitly tiled TIFF files showed dramatically different results on Azure VM. The external tiling approach significantly improves performance compared to the non-tiled Combined TIFF, but still does not match the performance of internally tiled options.

With Azure Data Lake, the internally tiled formats achieved the fastest format conversion and runtime performance. Azure Data Lake consistently outperforms mounted NAS for the benchmarking operations. For example, Combined tiled TIFF shows 25-40 second operation times on Data Lake versus 240-280 seconds on NAS.

Effects of Other Parameters

Setting predictor=2 significantly reduces file sizes within similar re-formatting time compared to predictor 1, highlighting that storage efficiency and processing speed can co-exist.

In terms of compression, DEFLATE and ZSTD show relatively small differences in benchmarking performance (typically within 10–20%), suggesting that for read-heavy workflows, re-formatting speed should take precedence over decompression efficiency.

3.4.2. Zarr Files

The Zarr specification is still evolving, and the benchmarks revealed that Zarr's performance generally lags behind the best-performing TIFF-based files.

Zarr's design utilizes an external tiling scheme, separating chunks into individual binary files and managing metadata via a hierarchical JSON file. While this architecture is conceptually straightforward, the external tiling resulted in slightly lower performance compared to internally tiled TIFF files.

Test Environments Specific Behavior

Zarr files stored in Azure Data Lake were nearly 2.5× faster than those in the mounted NAS. However, the fastest TIFF-based files achieved a speedup of 6× over the mounted NAS. This significant disparity suggests that the underlying infrastructure and tooling optimizations for Zarr are not yet as mature as those for the established GeoTIFF format in these production environments.

File Format	Blocksize	Compression	Predictor	File Size	Re-format (s)	10,000 Ops (s)
Combined TIFF	-	DEFLATE	1	38 GB	115.50	4216.61
			2	26 GB	184.03	3267.73
		ZSTD	1	38 GB	434.58	4313.61
			2	25 GB	266.60	3475.01
Combined tiled TIFF	256	DEFLATE	1	32 GB	**135.53**	**276.70**
			2	25 GB	**214.06**	**244.51**
		ZSTD	1	32 GB	416.50	366.14
			2	24 GB	292.42	273.75
Combined COG TIFF (no overview)	256	DEFLATE	1	32 GB	763.30	337.04
			2	25 GB	760.76	268.65
		ZSTD	1	33 GB	832.22	370.59
			2	25 GB	804.27	290.32
Combined COG TIFF	256	DEFLATE	1	44 GB	16527.25	283.19
			2	35 GB	16462.24	236.60
		ZSTD	1	44 GB	16207.74	325.17
			2	34 GB	16186.56	293.69
VRT + TIFF tiles	-	DEFLATE	1	36 GB	0.50 (default)	515.48
			2	30 GB	1121.31	491.14
		ZSTD	1	43 GB	1963.16	645.51
			2	26 GB	1886.93	510.56
VRT + tiled TIFF tiles	256	DEFLATE	1	36 GB	1068.26	403.64
			2	28 GB	1059.82	234.90
		ZSTD	1	35 GB	797.73	373.19
			2	26 GB	758.63	336.74
VRT + COG TIFF tiles (no overview)	256	DEFLATE	1	32 GB	690.34	334.36
			2	25 GB	729.15	265.96
		ZSTD	1	33 GB	813.50	417.42
			2	25 GB	792.28	371.08
VRT + COG TIFF tiles	256	DEFLATE	1	47 GB	20901.81	333.81
			2	35 GB	20998.18	279.19
		ZSTD	1	44 GB	20848.46	418.67
			2	35 GB	20938.46	346.94

Table 3.3: Performance comparison of different file formats ran on Azure VM with mounted NAS

File Format	Chunk size	Compression	File Size	Re-format (s)	10,000 Ops (s)
Zarr	(4, 256, 256)	GZIP (DEFLATE)	31 GB	4224.35	488.26
		ZSTD	32 GB	3703.37	545.30

Table 3.4: Performance comparison of Zarr configurations ran on Azure VM with mounted NAS

File Format	Blocksize	Compression	Predictor	File Size	Re-format (s)	10,000 Ops (s)
Combined TIFF	-	DEFLATE	1	38 GB	115.50	2999.02
			2	26 GB	184.03	2101.76
		ZSTD	1	38 GB	434.58	3129.64
			2	25 GB	266.60	2118.38
Combined tiled TIFF	256	DEFLATE	1	32 GB	**135.53**	**43.57**
			2	25 GB	**214.06**	**39.22**
		ZSTD	1	32 GB	416.50	43.83
			2	24 GB	292.42	38.93
Combined COG TIFF (no overview)	256	DEFLATE	1	32 GB	763.30	43.89
			2	25 GB	760.76	39.27
		ZSTD	1	33 GB	832.22	43.87
			2	25 GB	804.27	38.45
Combined COG TIFF	256	DEFLATE	1	44 GB	16527.25	48.30
			2	35 GB	16462.24	46.33
		ZSTD	1	44 GB	16207.74	48.24
			2	34 GB	16186.56	46.22
VRT + TIFF tiles	-	DEFLATE	1	36 GB	0.50 (default)	336.95
			2	30 GB	1121.31	289.90
		ZSTD	1	43 GB	1963.16	346.49
			2	26 GB	1886.93	289.62
VRT + tiled TIFF tiles	256	DEFLATE	1	36 GB	1068.26	82.52
			2	28 GB	1059.82	76.75
		ZSTD	1	35 GB	797.73	79.50
			2	26 GB	758.63	71.39
VRT + COG TIFF tiles (no overview)	256	DEFLATE	1	32 GB	690.34	83.06
			2	25 GB	729.15	74.80
		ZSTD	1	33 GB	813.50	85.94
			2	25 GB	792.28	77.40
VRT + COG TIFF tiles	256	DEFLATE	1	47 GB	20901.81	84.63
			2	35 GB	20998.18	79.42
		ZSTD	1	44 GB	20848.46	79.71
			2	35 GB	20938.46	76.47

Table 3.5: Performance comparison of different file formats ran on Azure VM and Azure Data Lake

File Format	Chunk size	Compression	File Size	Re-format (s)	10,000 Ops (s)
Zarr	(4, 256, 256)	GZIP (DEFLATE)	31 GB	4224.35	208.76
		ZSTD	32 GB	3703.37	211.07

Table 3.6: Performance comparison of Zarr configurations ran on Azure VM and Azure Data Lake

4

Conclusion

4.1. Task 1

This study presented a comprehensive evaluation of three computational approaches for a zonal statistics problem: the Vector-Raster method (baseline), the Raster-Raster method, and a Hybrid method.

4.1.1. Key Findings

Our initial hypothesis was confirmed. The Hybrid approach consistently matched or exceeded the performance of the better base method across all test cases, validating our assumption that intelligent method selection based on local occupancy characteristics can provide consistent improvements across diverse spatial data patterns.

Through exhaustive testing across multiple tiling schemes (from 1×1 to 32×32 subdivisions) and occupancy rate thresholds (0.02 to 0.08), we demonstrated that the optimal configuration varies significantly with spatial data characteristics. The Hybrid method achieved performance gains of up to 733% compared to the baseline method on dense tiles like `18_42`, while maintaining comparable efficiency on sparse tiles.

However, the most significant and unexpected finding was that spatial tiling itself provides substantial performance improvements that often exceed the gains from algorithmic hybridization. For instance, the Vector-Raster baseline method improved from 985 seconds without tiling to 216 seconds with 8×8 tiling on tile `18_42`—a 4.5$\times$ speedup. This improvement stems from both the ability to skip sparse regions.

4.1.2. Limitations

Several limitations should be marked. First, our evaluation was conducted on a limited set of four representative tiles, which may not capture the full spectrum of spatial data characteristics. Second, the memory constraints that necessitated tiling for the Raster-Raster method may have influenced the comparative results, potentially masking the true performance characteristics of each algorithm when unconstrained by memory limitations. Third, the occupancy rate threshold optimization revealed that optimal parameters are highly data-dependent, suggesting that a fixed threshold may not be suitable for heterogeneous datasets requiring automated processing pipelines.

4.1.3. Future Work

Three unexplored directions emerge from this work. First, developing adaptive threshold selection mechanisms that can automatically determine optimal occupancy rate parameters would enhance the applicability of the Hybrid method. Second, exploring alternative spatial partitioning strategies beyond regular grid tiling—such as adaptive quadtree decomposition or DBSCAN [5]—could potentially yield further performance improvements. Third, investigating the scalability of these approaches to larger datasets and different spatial resolutions could provide more insights.

4.2. Task 2

4.2.1. Key Findings

The experiments demonstrate that the choice of file format and storage backend has a profound impact on I/O performance for the simulated benchmarking task. The **Combined Tiled TIFF** format is concluded to provide the optimal balance of fast re-formatting time and superior read performance. The most critical factor influencing performance was the storage environment. **Azure Data Lake** massively outperformed the mounted NAS, yielding a $6 - 7\times$ speedup for the optimal TIFF format.

The results also show that **internal tiling** is essential for efficient data access. The non-tiled "Combined TIFF" format was catastrophically slow, while external tiling via VRT was an improvement but still inferior to a internally-tiled file.

Interestingly, the additional overhead of Cloud-Optimized GeoTIFFs (COGs), specifically their raster pyramids, was confirmed to be unnecessary for the specific use case and incurred a long re-formatting time. Finally, the Zarr format, in its tested configuration, lagged behind the optimized TIFF formats, suggesting that its tooling or I/O patterns are not as mature for Readar's production environment.

4.2.2. Limitations

This study's primary limitation is the **absence of a GPU** in the benchmarking environment. While the experiments were designed to simulate the data-loading bottlenecks of deep learning workflows, they only measured the "storage-to-memory" (CPU) portion of the task. This omits the critical step of data transfer and decompression into GPU memory.

Modern data-loading pipelines increasingly bypass the CPU for these tasks. For instance, the Zarr community has developing support for direct-to-GPU integration[1], and specialized libraries like NVIDIA's nvTIFF[2] exist to accelerate GeoTIFF access.

Furthermore, the scope of this study was limited even within the CPU-based environment. A wide array of modern data-access libraries were not evaluated. These include libraries designed for asynchronous I/O (e.g., `async-tiff`[3] and `aiocogeo`[4]), and other spatial analysis tools (e.g., `xarray-spatial`[5]).

4.2.3. Future Work

Based on the limitations identified, the following are proposed for future research:

- **Benchmark GPU pipelines:** This involves measuring the end-to-end time from file read to a tensor resident in GPU memory. This work should explicitly evaluate the performance of specialized libraries for GeoTIFFs and Zarr.
- **Evaluate new, modern libraries:** A comparative benchmark should be conducted on the untested CPU-based libraries. Evaluating asynchronous tools (e.g., `async-tiff`) will determine if they can offer a more performant way for CPU-based data loading.

[1] https://zarr.readthedocs.io/en/stable/user-guide/gpu.html
[2] https://docs.nvidia.com/cuda/nvtiff
[3] github.com/developmentseed/async-tiff
[4] github.com/geospatial-jeff/aiocogeo
[5] xarray-spatial.readthedocs.io

5

Personal Reflection

My knowledge and skills were just sufficient to get me into this internship position and start learning new things. All of my work will be used in a production environment; this work is not merely a proof-of-concept. We have to ensure that all the calculation results are logically correct. I therefore delved into many different packages' documentation, e.g., GeoPandas[1], xarray[2] and rioxarray[3], Dask[4], and Icechunk[5].

During the internship, I sometimes found myself lost in the tasks and was uncertain about what to do. I have to make the projects somewhat academic, but I also have to pay attention to the software implementation details, like robustness, user experience (for GUI and CLI application design), and function designs (for readability and reusability). Despite having some doubts, I'd still say I learned a lot. I knew Rasterio and GDAL, but I never had to look into their implementation details, in terms of the algorithms, runtime, and the resources they use. This is the first time I've worked with GDAL VSI, Zarr, ZOG, Xarray, Icechunk, and even GeoPandas. It is also the first time I've used SQLAlchemy together with PostGIS to write geometry data into a database. Some of the knowledge can only be learned in a production environment that maintains large-scale data.

I had the most fun doing Tasks 1 and 3 because they allowed me to approach the entire problem and develop original solutions. In contrast, Task 2 was a significant experiment that took up most of my time and presented the highest number of issues and challenges. If I had more time, I'd try to read more literature and take a step back from immediately trying to solve the problems on my own. While it was rewarding to solve problems using my own creativity, I recognize that there is a whole community of researchers who may already have better solutions.

Lastly, it's encouraging to see that many of the modern, cloud-optimized formats have been included in GEO5019 (Geomatics Studio) this year. I believe this is a valuable addition, as massive file size is inherent to the nature of geo-spatial data.

[1] https://geopandas.org
[2] https://xarray.dev
[3] https://corteva.github.io/rioxarray
[4] https://docs.dask.org
[5] https://icechunk.io

References

[1] Open Geospatial Consortium et al. *Zarr Storage Specification 2.0 Community Standard*. 2023.

[2] Ahmed Eldawy et al. "Large scale analytics of vector+ raster big spatial data". In: *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 2017, pp. 1–4.

[3] David Haynes, Steve Manson, and Eric Shook. "Terra Populus' architecture for integrated big geospatial services". In: *Transactions in GIS* 21.3 (2017), pp. 546–559. DOI: https://doi.org/10.1111/tgis.12286. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/tgis.12286. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/tgis.12286.

[4] Open Geospatial Consortium. *OGC GeoTIFF Standard*. Standard 19-008r4. Version 1.1. Accessed: 2025-05-12. Open Geospatial Consortium, 2019. URL: https://www.ogc.org/standards/geotiff/.

[5] Erich Schubert et al. "DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN". In: 42.3 (July 2017). ISSN: 0362-5915. DOI: 10.1145/3068335. URL: https://doi.org/10.1145/3068335.

[6] Samriddhi Singla and Ahmed Eldawy. "Distributed zonal statistics of big raster and vector data". In: SIGSPATIAL '18. Seattle, Washington: Association for Computing Machinery, 2018, pp. 536–539. ISBN: 9781450358897. DOI: 10.1145/3274895.3274985. URL: https://doi.org/10.1145/3274895.3274985.

[7] Samriddhi Singla and Ahmed Eldawy. "Raptor Zonal Statistics: Fully Distributed Zonal Statistics of Big Raster + Vector Data". In: *2020 IEEE International Conference on Big Data (Big Data)*. 2020, pp. 571–580. DOI: 10.1109/BigData50022.2020.9377907.

[8] Adam J. Stewart et al. "TorchGeo: Deep Learning With Geospatial Data". In: *ACM Transactions on Spatial Algorithms and Systems* (Dec. 2024). DOI: 10.1145/3707459. URL: https://doi.org/10.1145/3707459.

[9] Akram Zaytar et al. *Optimizing Cloud-to-GPU Throughput for Deep Learning With Earth Observation Data*. 2025. arXiv: 2506.06235 [cs.CV]. URL: https://arxiv.org/abs/2506.06235.

[10] Jianting Zhang, Simin You, and Le Gruenwald. "Efficient Parallel Zonal Statistics on Large-Scale Global Biodiversity Data on GPUs". In: *Proceedings of the 4th International ACM SIGSPATIAL Workshop on Analytics for Big Geospatial Data*. BigSpatial '15. Bellevue, WA, USA: Association for Computing Machinery, 2015, pp. 35–44. ISBN: 9781450339742. DOI: 10.1145/2835185.2835187. URL: https://doi.org/10.1145/2835185.2835187.

A

Task 3: Application for Feature Creation and QC

Task 3 is focused on software implementation rather than experiment. The core objective was to replace a part of the existing stereo matching workflow—an obsolete MATLAB script—with a modern Python application that includes enhanced functionality. While the task was conceptually simple, the reliance on an outdated MATLAB script made the legacy process a severe operational bottleneck for the company.

The outcome is a comprehensive application that provides a graphical user interface (GUI) to orchestrate a complex geo-spatial data pipeline. This application guides the user through a sequential workflow of data loading, validation, footprint calculation, and precise write operations (insert, update, upsert) to a database, while offering real-time data inspection and flexible configuration management.

A.1. Application Workflow

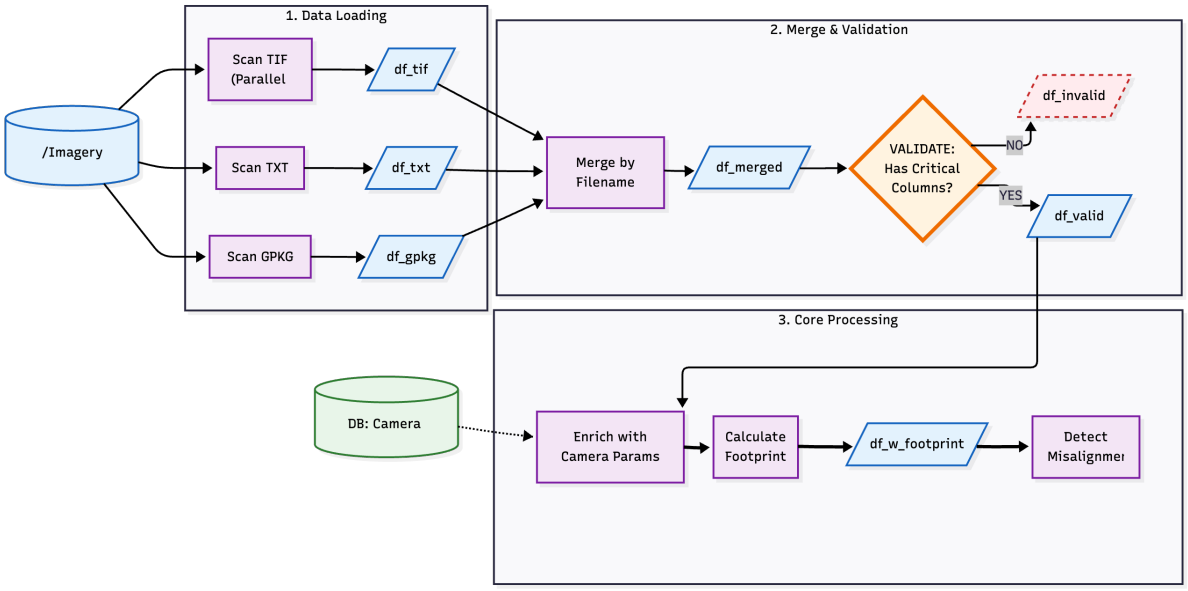


Figure A.1: Application workflow

The application achieves its objectives through the following sequential workflow steps:

1. **Data Extraction:** Parallel processing of TIF metadata, TXT, and GPKG files is performed from local or cloud storage via GDAL VSI[1]. E.g., mounted NAS or Azure Data Lake.

[1]VSI (Virtual System Interface): https://gdal.org/en/stable/user/virtual_file_systems.html

2. **Data Validation & Processing:** This involves merging, type casting, value corrections, and enriching the data with necessary camera parameters.
3. **Footprint Calculation:** The system computes image footprints on the ground surface using camera intrinsics and DTM data, supporting both sequential and parallel implementations.
4. **Misalignment Detection:** Identifies images with footprint misalignment (optional).
5. **Database Operations:** Executes insertion, updating, and querying operations on the `PostgreSQL` database, including geometry support (optional).

A.2. Data

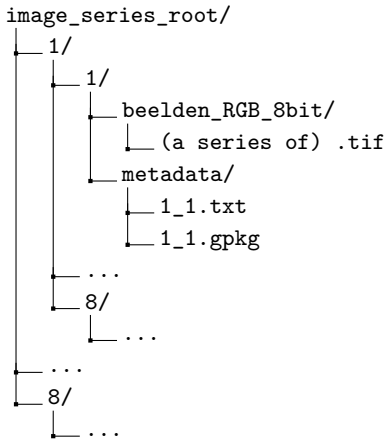


Figure A.2: Representative image series folder structure

As shown in Figure A.2, the dataset employs a hierarchical structure, this nested structure allows for more than 40 distinct image series and a total of around $300,000$ TIFF files. The root level contains primary numerical directories (1 through 8 in this case), each representing a major grouping. Inside each primary directory is a second level of subdirectories, which function as individual image series containers. Finally, within every second-level container, there are two terminal directories: `beelden_RGB_8bit/`, which presumably holds the raw image files, and `metadata/`, which stores associated descriptive information like TXT and GPKG files.

TIFF files contain primary information such as filename, path, datetime, and camera info (if available). TXT files provide extrinsic orientations (e.g., the position and orientation of the camera). If metadata is missing from the TIFF or TXT files, GPKG files serve as a source for supplementary information.

A.3. Results

In this section, the functionality of the resulting GUI application (and hence CLI application) will be explained. As shown in Figure A.3 and A.4, the app consists of five major blocks, i.e., **Required Parameters**, **Data Processing Steps**, **Database Operations**, **Data Viewer** and **Configuration**.

A.3.1. Required Parameters

- **Database parameters**: This is necessary for steps like **Merge and Validate** and **Calculate footprint**, as the program needs to fetch related data from the database.
- **Azure parameters**: This is needed if the user wants to utilize GDAL VSI for the faster loading speed (roughly $2\times$ faster compare to a mounted NAS).
- **Imagery directory**: The path of the image series to be scanned.
- **Image folder name**: The folder name which stores the TIFF images.
- **Metadata folder name**: The folder name which stores the metadata (i.e., TXT and GPKG files).

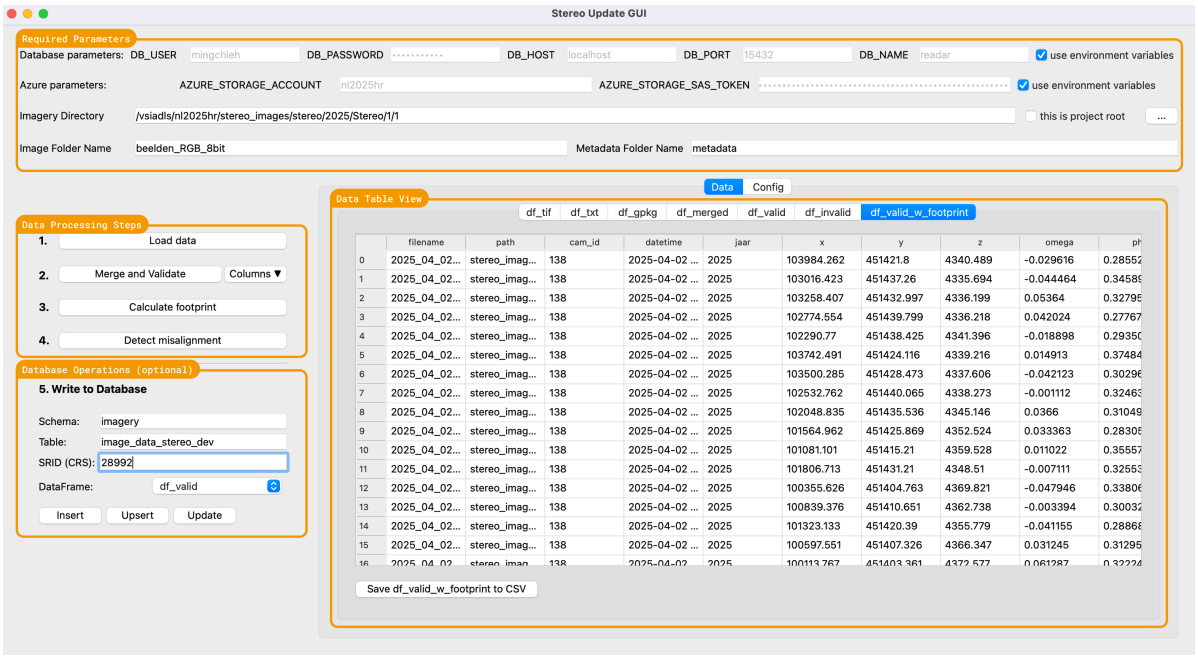


Figure A.3: Application GUI (showing data view)

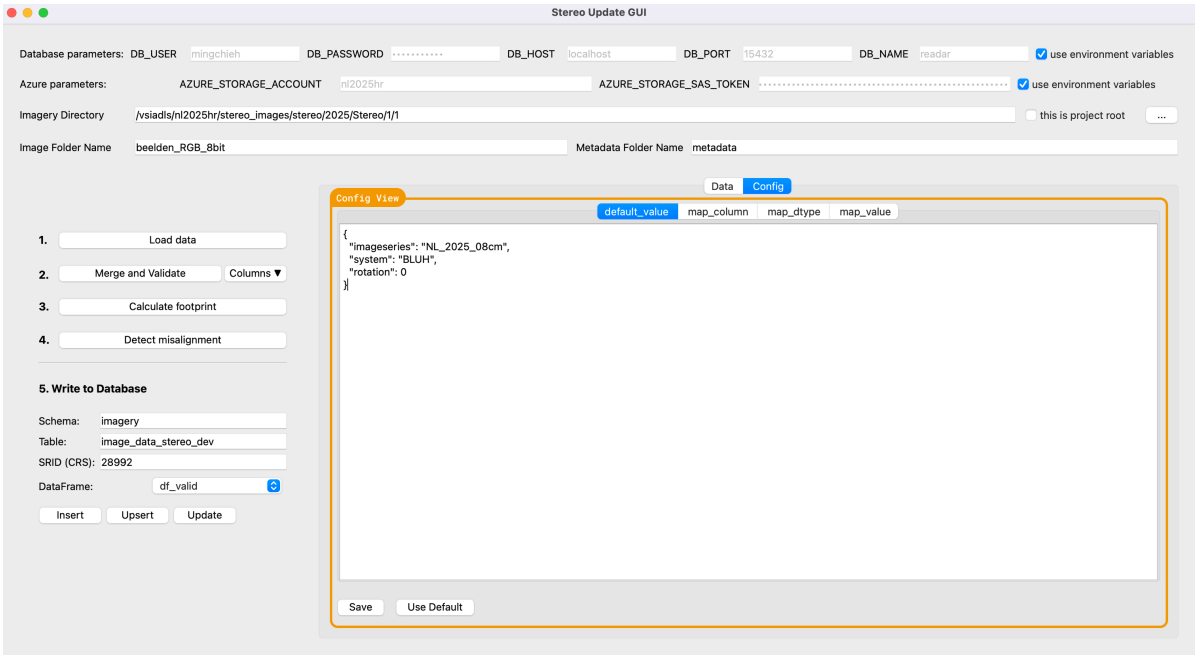


Figure A.4: Application GUI (showing config view)

A.3.2. Data Viewer

There are seven data tabs (i.e., `df_tif`, `df_txt`, `df_gpkg`, `df_merged`, `df_valid` and `df_invalid`, `df_valid_w_footprint`) in the data viewer, each represents the outcome of the data processing steps, detailed in Data Processing Steps. For each table there is an option to save the data to a user assigned location.

A.3.3. Data Processing Steps

- **Load data**: Scan the `Imagery directory` and load all the TIFF, TXT and GPKG files from it. This step will generate three tables, namely `df_tif`, `df_txt` and `df_gpkg` also does some correction to prepare the data for merging. For fetching the TIFF files, despite we are only interested in the image metadata in their header, there is too much I/O overhead to deal with, hence parallel processing is used to fetch the TIFF files' metadata.
- **Merge and Validate**: First, merge the three tables fetched from last step, using filename as foreign key, creating `df_merged`. Secondly, validate if the rows in `df_merged` contain all the necessary columns needed for calculating footprint. The validation step will generate two tables, `df_valid` and `df_invalid`, only the `df_valid` will be used from the next step, the `df_invalid` will be displayed in the data viewer for inspection.
- **Calculate footprint**: This step will first enrich the `df_valid` table using camera parameters fetched from database, and then perform the footprint calculation. Every valid data row will receive a footprint as the program has made sure these rows are not missing the critical column for calculating footprint.
- **Detect misalignment**: After the footprint calculation is done, the footprints will be examined by checking the y-offset per flight line.

A.3.4. Database Operations

There are three ways to write the processed data to a database: insert (ignore on conflict), upsert (update on conflict) and update (update existing rows with the same filename without inserting new data rows).

- **Schema**: Name of the schema
- **Table**: Name of the table
- **SRID (CRS)**: The EPSG code of the coordinate reference system (e.g., SRID 4326 for WGS84), this is used to create PostGIS geometries in the database.
- **Dataframe**: Users can select which table they want to write to database, with two options, `df_valid` or `df_valid_w_footprint`.

A.3.5. Configuration

Figure A.4 displays the "config" tab, which contains four tabs, each with a text editor block. The configurations are given in JSON format, the usage of each is explained below.

- `default_value`: Provide the default values to fill in when null
- `map_column`: Repair potential typographical errors in column names, or can merge multiple columns with semantic incompatibility (same data format but with different column names).
- `map_dtype`: Mapping all columns to correct data type.
- `map_value`: Repair incorrect, corrupted values using a 1-to-1 mapping.

B

Hardware and Software Specifications

B.1. Personal Laptop

Component	Specification
Device	MacBook Pro, 14-inch (2021)
CPU	Apple M1 Pro 8-core (6 performance + 2 efficiency)
Memory Bandwidth	200 GB/s
Memory	16 GB unified memory
Operating System	macOS Sonoma 14.7.1
Python Version	3.12.7
GADL Version	3.11.0 ”Eganville”

Table B.1: Hardware and software specification

B.2. Azure VM

Component	Specification
Device	Azure VM
CPU	Intel(R) Core(TM) i7-14700K 28-core
Memory Bandwidth	90 GB/s
Memory	64 GB
Operating System	Ubuntu 22.04.5 LTS (x86_64)
Python Version	3.12.7
GADL Version	3.11.0 ”Eganville”

Table B.2: Hardware and software specification

C

Additional Information for Task 1

C.1. Comprehensive Evaluation: Detailed Plots

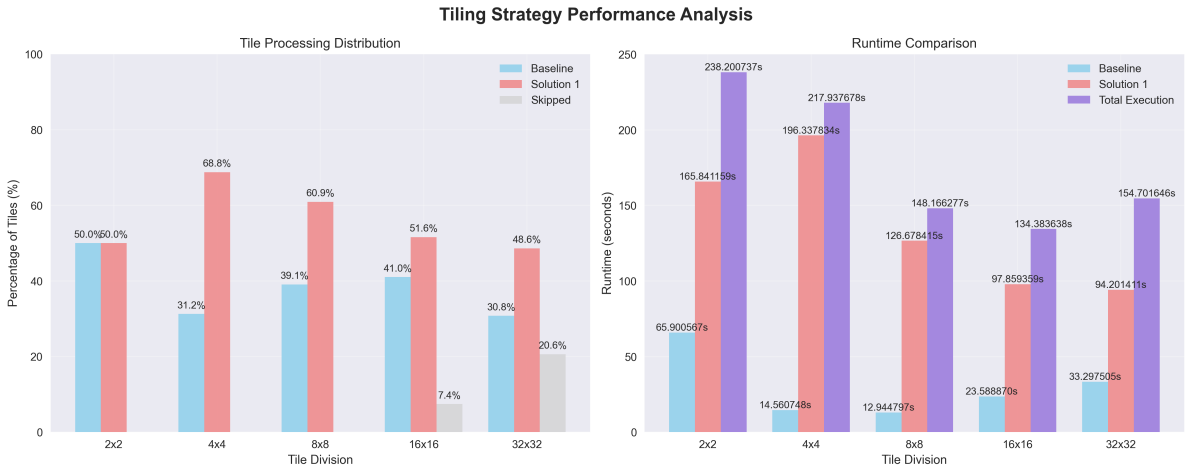


Figure C.1: Tile division analysis of tile 18_42

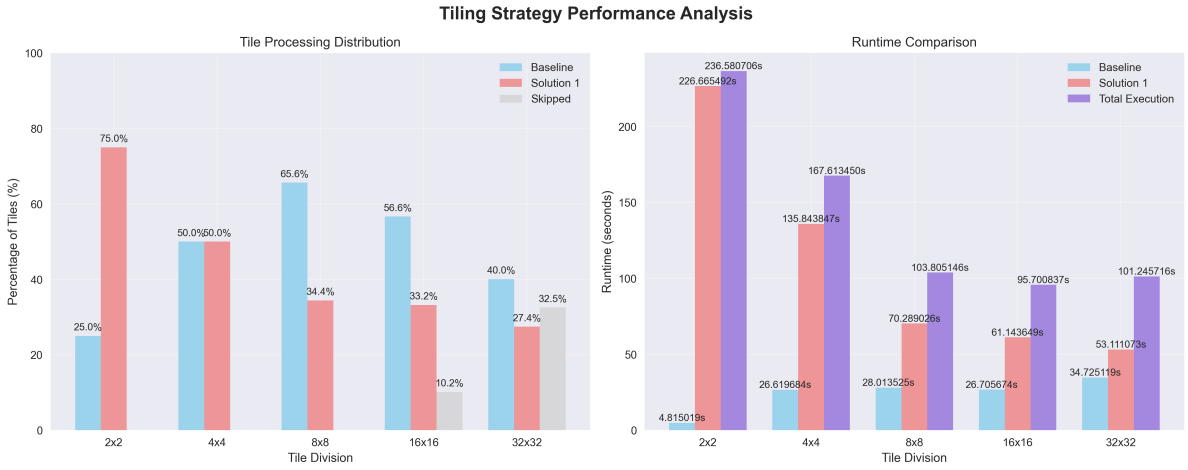


Figure C.2: Tile division analysis of tile 18_43

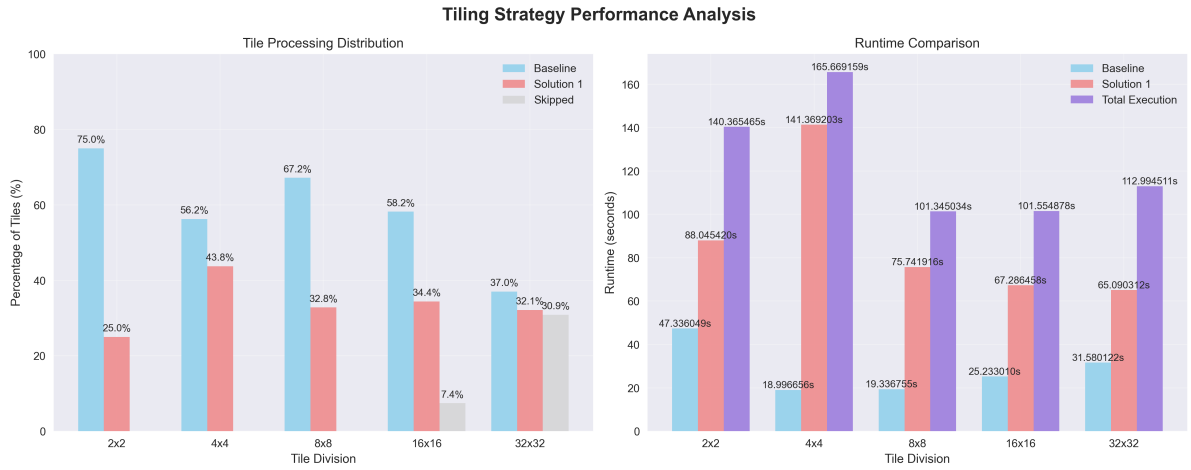


Figure C.3: Tile division analysis of tile 20_50

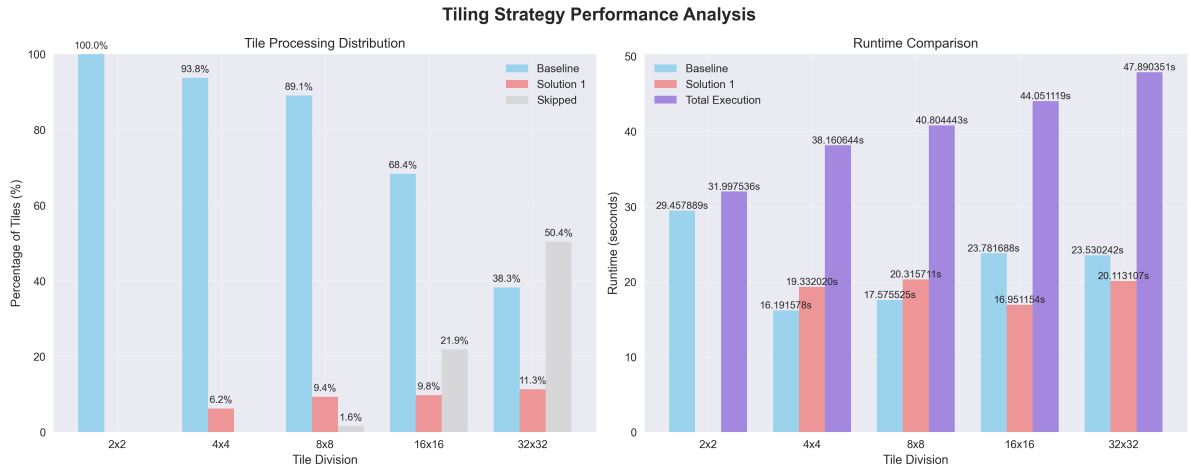


Figure C.4: Tile division analysis of tile 24_52

C.2. Evaluation Across Different Thresholds: Detailed Plots

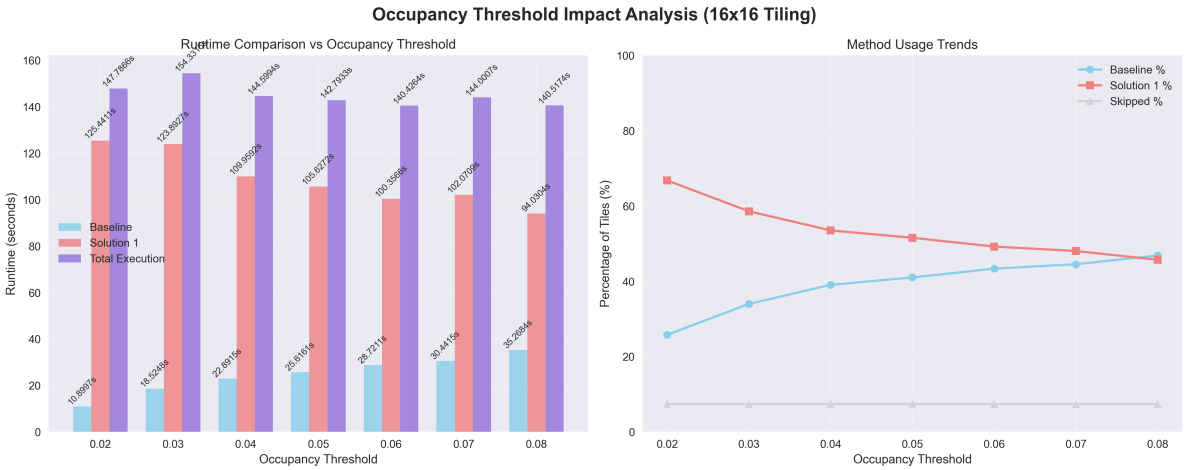


Figure C.5: $OccupancyRate$ threshold analysis of tile 18_42

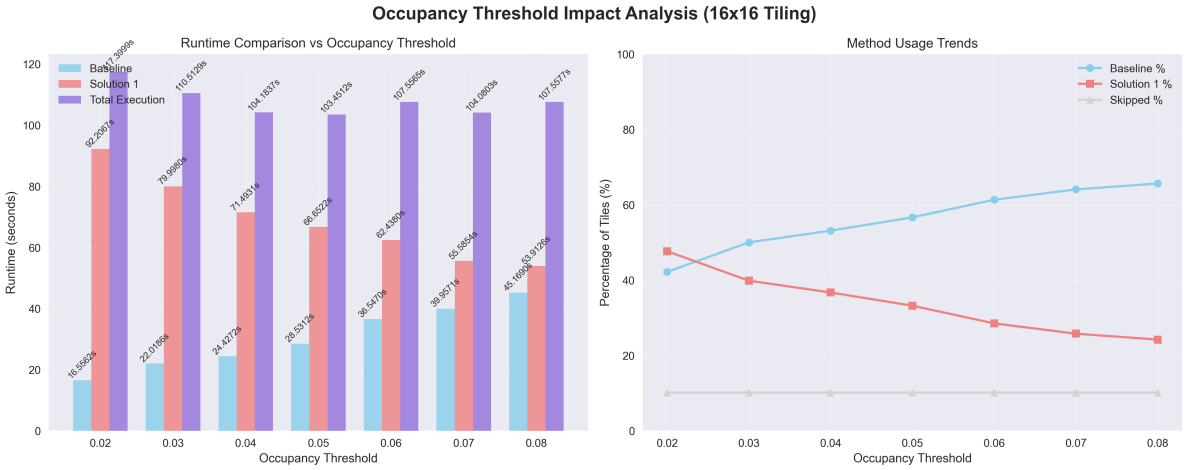


Figure C.6: $OccupancyRate$ threshold analysis of tile 18_43

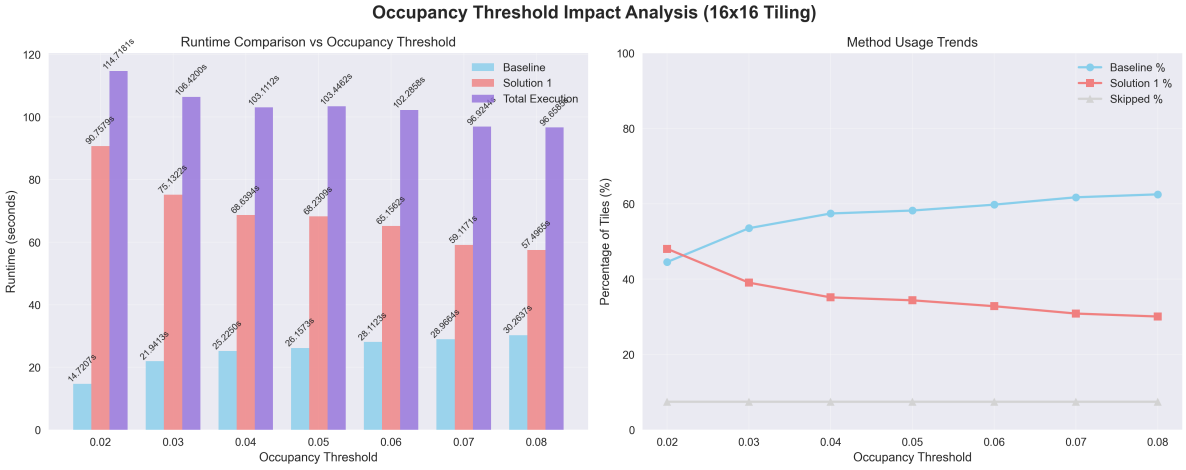


Figure C.7: $OccupancyRate$ threshold analysis of tile 20_50

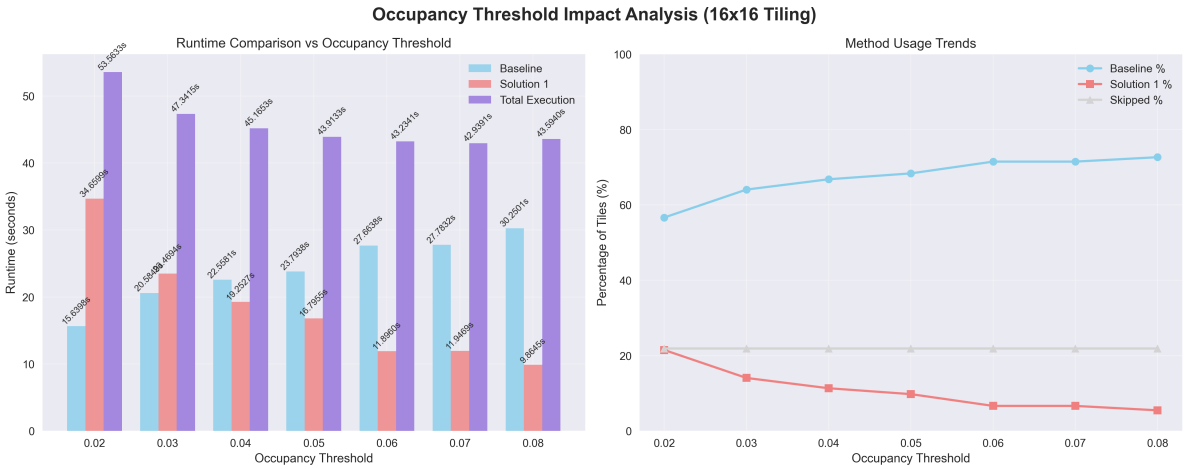


Figure C.8: $OccupancyRate$ threshold analysis of tile `24_52`

D

Additional Information for Task 2

D.1. Software Implementation for Handling TIFF-based Files

Combining TIFF-based Files

`temp_vrt` (str): Path to the temporary VRT (Virtual Raster) file that serves as the source for creating the raster variant. This file references the original raster data without duplicating it.

`output_path` (str): File path where the new raster variant will be saved. This includes the full path and filename for the output raster file.

`variant_name` (str): Descriptive name for the raster variant used for logging.

`format_type` (str): GDAL format driver identifier specifying the output format. Common values include `'GTiff'` or `'COG'`.

`creation_options` (list): List of GDAL creation options that control format-specific parameters such as:

- Compression algorithms (e.g., `'COMPRESS=DEFLATE'`)
- Tiling specifications (e.g., `'TILED=YES'`)
- Block sizes (e.g., `'BLOCKXSIZE=256'`)
- Predictor settings (e.g., `'PREDICTOR=2'`)

```python
def make_combine_raster(temp_vrt, output_path, variant_name, format_type, creation_options):
    translate_options = gdal.TranslateOptions(
        format=format_type,
        creationOptions=creation_options
    )

    result_ds = gdal.Translate(
        str(output_path),
        str(temp_vrt),
        options=translate_options
    )

    if result_ds is None:
        return None

    result_ds = None  # Close dataset
    return {
        'variant': variant_name,
        'format': format_type,
        'output_path': str(output_path),
    }
```

Listing D.1: Function to combine TIFF sub-tiles into one

Converting TIFF-based Files Without Combining

`input_file` (str or Path): Path to the source raster file that will be converted to a different format variant.

`variant` (dict): Dictionary containing the configuration parameters for the raster variant. Including:

- `'name'`: String identifier for the variant (used for folder naming)
- `'format'`: GDAL format driver (e.g., `'GTiff'`, `'COG'`)
- `'options'`: List of GDAL creation options for format-specific parameters

`output_dir` (Path): Base directory where the converted raster variants will be stored. A subdirectory will be created for each variant based on the variant name.

```python
def convert_raster(input_file, variant, output_dir):
    input_path = Path(input_file)
    base_name = input_path.stem

    variant_folder = output_dir / variant['name']
    output_file = variant_folder / f"{base_name}.tif"

    translate_options = gdal.TranslateOptions(
        format=variant['format'],
        creationOptions=variant['options']
    )

    ds = gdal.Translate(
        str(output_file),
        str(input_file),
        options=translate_options
    )

    if ds:
        ds = None  # Close dataset
        return {
            'input_file': input_path.name,
            'variant': variant['name'],
            'format': variant['format'],
            'processing_time_seconds': processing_time,
            'file_size_gb': file_size_gb,
            'output_path': str(output_file),
            'variant_folder': variant_folder
        }
    else:
        return None
```

Listing D.2: Function to combine TIFF sub-tiles into one

Clipping Function for TIFF-based Files

The function shown in Listing D.3 is designed to be called in parallel using Dask, where each row represents one polygon feature from a vector dataset, and the same `raster_path` and `output_dir` are used for all polygons in the processing batch.

`raster_path` (str): File path to the source raster dataset that will be clipped. This is the large input raster file in one of your experimental formats (TIFF, COG or VRT).

`row` (GeoDataFrame row): A single row from a GeoPandas DataFrame containing polygon geometry and attributes. Must include:

- `'wkb_geometry'`: The polygon geometry used for clipping the raster
- `'identificatie'`: Unique identifier used for naming the output file

`output_dir` (str): Directory path where the clipped raster files will be saved. Each polygon will generate a separate GeoTIFF file in this directory.

```python
@delayed    # Parallel process with geopandas dask
def task_tiff(raster_path, row, output_dir: str):
    with rasterio.open(raster_path) as raster:
        polygon = row['wkb_geometry']
        geojson_polygon = [mapping(polygon)]
        nodata_value = raster.nodata

        # Clip the raster with the polygon
        try:
            out_image, out_transform = mask(raster, geojson_polygon, nodata=nodata_value, crop=True)
        except Exception as e:
            print(e)
            print(row['identificatie'])

            return None

        # Update metadata for the clipped raster
        profile = raster.profile.copy()
        profile.update({
            'count': out_image.shape[0],
            'height': out_image.shape[1],
            'width': out_image.shape[2],
            'transform': out_transform,
            'dtype': out_image.dtype,
            'nodata': nodata_value
        })

        # Write the new raster
        output_path = os.path.join(output_dir, f"{row['identificatie']}.tif")
        with rasterio.open(output_path, 'w', **profile) as dst:
            dst.write(out_image)
```

Listing D.3: Function to perform the clipping task in TIFF-based format

D.2. Software Implementation for Handling Zarr Files

Zarr is a data standard with multiple implementations maintained by different organizations (e.g., the official `zarr` library, `xarray`, and the GDAL Zarr driver). While all produce valid Zarr files, their outputs differ slightly in structure and metadata. For creating the array with most compatibility across all libraries, we use the official `zarr` library, as shown in Section D.2.

Zarr is suitable for array-like data storage and also enables fast random access, but it is not designed specifically for geo-referenced data. As a result, additional steps are required to attach coordinate information needed for clipping operations (see Section D.2). Currently, only `xarray`[1] supports complete clipping functionality.

In addition, Zarr relies on multi-threaded compression libraries (such as Blosc), which interact with Python's Global Interpreter Lock (GIL). This affects how parallelism can be leveraged during benchmarking. To ensure a fair comparison with TIFF-based formats, the Zarr benchmarking task required an alternative implementation (see Section D.2 and D.2).

Converting Files to Zarr Format

```python
def create_empty_zarr_store(output_path, extent_info, chunk_size=(4, 256, 256)):
    """Create empty Zarr group with imagery array and metadata for streaming data later"""

    if os.path.exists(output_path):
        print(f"Removing existing zarr store: {output_path}")
        shutil.rmtree(output_path)

    # Get dimensions
    bands = extent_info['bands']
    total_height, total_width = extent_info['shape']
    shape = (bands, total_height, total_width)
    chunks = (chunk_size[0], chunk_size[1], chunk_size[2])

    # Create zarr group
    zarr_group = zarr.open_group(output_path, mode='w')

    # Add group-level metadata
    zarr_group.attrs.update({
        'bounds': extent_info['bounds'],
        'pixel_size': extent_info['pixel_size'],
        'crs': 'EPSG:28992',
        'description': 'Geospatial imagery dataset'
    })

    # Create imagery array within the group
    zarr_array = zarr.create_array(
        store=zarr_group.store,
        name='imagery',  # name within the group
        shape=shape,
        chunks=chunks,
        dtype=np.uint8,
        compressors=[zarr.codecs.BloscCodec()],
        fill_value=0,
        zarr_format=3,
        dimension_names=['band', 'y', 'x']
    )

    # Consolidate metadata
    zarr.consolidate_metadata(output_path)

    print(f"Created Zarr group with imagery array: shape={zarr_array.shape}, dtype={zarr_array.dtype}")
    return zarr_array
```

Listing D.4: Function to create an empty Zarr storage with metadata

```python
def stream_tifs_to_simple_zarr(zarr_array, extent_info):
    """Stream TIF data to zarr array"""

```

[1]Xarray: https://docs.xarray.dev/en/stable

```
    global_bounds = extent_info['bounds']
    pixel_size_x, pixel_size_y = extent_info['pixel_size']
    total_height, total_width = extent_info['shape']

    for file_info in extent_info['files']:
        print(f"Processing {os.path.basename(file_info['file'])}...")

        with rasterio.open(file_info['file']) as src:
            file_bounds = file_info['bounds']

            # Calculate position
            col_start = int((file_bounds.left - global_bounds[0]) / pixel_size_x)
            row_start = int((global_bounds[3] - file_bounds.top) / pixel_size_y)

            # Process each band
            for band in range(file_info['count']):
                band_data = src.read(band + 1)

                row_end = row_start + band_data.shape[0]
                col_end = col_start + band_data.shape[1]
                row_end = min(row_end, total_height)
                col_end = min(col_end, total_width)

                if row_start >= 0 and col_start >= 0:
                    zarr_array[band, row_start:row_end, col_start:col_end] = \
                    band_data[:row_end-row_start, :col_end-col_start]

                del band_data

            print(f"[{row_start}:{row_end}, {col_start}:{col_end}]")
```

Listing D.5: Function to stream the TIFF files to Zarr storage

Clipping Function for Zarr Files

`zarr_path` (str): File path to the source Zarr dataset that will be clipped.

`row` (GeoDataFrame row): A single row from a GeoPandas DataFrame containing polygon geometry and attributes. Must include:

- `'wkb_geometry'`: The polygon geometry used for clipping the raster
- `'identificatie'`: Unique identifier used for naming the output file

`output_dir` (str): Directory path where the clipped raster files will be saved. Each polygon will generate a separate GeoTIFF file in this directory.

```
@delayed    # Parallel process with geopandas dask
def task_zarr(zarr_path, row, output_dir: str):
    # Load ZARR, add necessary info, assign coordinates to raster
    ds = xr.open_zarr(zarr_path)
    ds = ds.rio.write_crs("EPSG:28992")
    ds['imagery'] = ds['imagery'].rio.write_nodata(0)
    bounds = ds.attrs['bounds']
    pixel_size = ds.attrs['pixel_size'][0]
    x_coords = np.arange(bounds[0], bounds[2], pixel_size)
    y_coords = np.arange(bounds[3], bounds[1], -pixel_size)
    ds['imagery'] = ds['imagery'].assign_coords({
        'x': ('x', x_coords[:ds['imagery'].shape[1]]),
        'y': ('y', y_coords[:ds['imagery'].shape[0]])
    })

    # Get polygon bounds for bbox clipping
    polygon = row['wkb_geometry']
    bounds = polygon.bounds  # (minx, miny, maxx, maxy)

    try: # Clip by bounding box first (fast, reduces data read)
        imagery = ds["imagery"].rio.clip_box(*bounds, crs=ds.rio.crs)
    except Exception as e:
        print(e)
        print(row['identificatie'])
```

```
        return None

    # Then clip by polygon (still lazy, uses Dask)
    clipped = imagery.rio.clip([polygon], crs=ds.rio.crs, from_disk=True)
    output_path = os.path.join(output_dir, f"{row['identificatie']}.tif")
    clipped = clipped.transpose('band', 'y', 'x')
    clipped.rio.to_raster(output_path)
```

Listing D.6: Function to perform the clipping task in Zarr

Clipping in parallel for Zarr Files

```
# ...
# gdf is the geometries loaded from database
ddf = dask_geopandas.from_geopandas(gdf, npartitions=16)
tasks = [task_zarr(ZARR_PATH, row, IMG_DIR) for _, row in tqdm(ddf.iterrows(), total=len(ddf), desc="
    Allocating Tasks")]

with ProgressBar():
    dask.compute(*tasks, scheduler='processes', num_workers=16)
```

Listing D.7: Script to benchmark Zarr file with multi-process instead of multi-threads

Experimenting on Benchmarking Zarr in Parallel

According to the Zarr documentation, it is possible to read Zarr files in parallel. However, several developers[2] have shared experimental results indicating that parallel reading does not significantly improve performance. In another post[3], one user noted that "Zarr already uses multiple threads during decompression. Blosc usually does a good job of making use of available cores, so you may not be able to improve much, if at all, by adding further multi-threading." Our experiment showed similar results, as summarized below in Table D.1.

File Format	Chunk size	Compression	File Size	Re-format (s)	10,000 Ops (s)
Zarr	(4, 256, 256)	ZSTD	32 GB	1876.60	217.72 (multi-processing) 855.24 (multi-threading) 845.19 (run in sequential)

Table D.1: Performance evaluation on PC: multi-threading versus sequential processing

[2]Discussion on GitHub: https://github.com/zarr-developers/zarr-python/discussions/2184
[3]Discussion on StackOverflow:https://stackoverflow.com/questions/56781085/zarr-multithreaded-reading-of-groups