Universiteit
Leiden
The Netherlands

Designing a Reproducible AI Pipeline for Automated Image Analysis in OMERO

Sacha Northolt (s3697711)

Supervisors:
Dr. L. Cao & Dr. M.W. Paul

Bachelor Thesis– Bioinformatica

Leiden Institute of Advanced Computer Science (LIACS)
liacs.leidenuniv.nl

June 29, 2026

Abstract

Microscopy research is generating ever-increasing amounts of image data, but tools for automated analysis are often not directly integrated with the platforms where that data is stored. OMERO is a widely used platform for managing biological image data, but offers no built-in support for AI-based image analysis. In this thesis, a modular and reproducible AI pipeline is built that links OMERO to existing deep learning frameworks. The pipeline is implemented as a Jupyter notebook and utilises BiaPy with the affable-shark model and StarDist with the `2D_versatile_fluo` model for instance segmentation of cell nuclei. Images are automatically retrieved from OMERO, processed and segmented, after which the results, including metadata, are written back to OMERO. Switching between models requires only a single line change, though a kernel restart is currently required between models. StarDist performed better on the FUCCI dataset than BiaPy due to the densely packed cell nuclei in that dataset. The pipeline demonstrates that modular and reproducible AI integration within OMERO is feasible, without requiring expertise in deep learning implementation.

Keywords: OMERO, deep learning, instance segmentation, BiaPy, StarDist, bioimaging, FAIR data, cell segmentation

Contents

1 Introduction

Advances in microscopy techniques mean that biological studies are generating ever-increasing amounts of image data. Modern high-throughput microscopy studies can generate thousands to hundreds of thousands of images per experiment, for example in high-content screening (HCS), where large numbers of conditions are photographed automatically[6]. As these datasets grow, manual analysis has become less scalable and more susceptible to subjective differences between researchers. Reproducibility in biology is becoming an increasingly significant problem. Manual annotation of cells is time-consuming and varies from researcher to researcher [29]. For this reason, the automation of image analysis is becoming increasingly important. The intended users are biologists and biomedical researchers who work with microscopy data on a daily basis. Although they possess biological knowledge, most researchers have no programming experience and are unable to write code or configure complex software independently. At present, they have to analyse images manually, which is slow and subjective, or rely on a programmer for help, creating an unwanted dependency. The aim of this thesis is to bridge that gap.

1.1 OMERO as Data Management Platform

One option for storing microscopic data is OMERO. This is an open-source platform where biological data is uploaded and stored by researchers from all over the world[5]. OMERO supports a wide range of microscopic file formats and associated metadata. Researchers can view, annotate and share images via a central server. Leiden University was one of the first in the Netherlands to adopt OMERO. Since 2020, all researchers here have been required to upload their data to the platform[16]. As a result, OMERO has become an essential part of the data infrastructure within the faculty. The advantage of OMERO is that metadata can be added to each image, in addition to annotations and experimental information[17]. This makes the platform very useful in biological and medical research. Within the university's central server, images can be easily shared between different research groups. This makes large datasets more manageable. However, the system is not designed to perform analyses. Tasks such as cell segmentation, object detection and classification cannot be carried out directly within the platform. At present, researchers have to perform analyses manually or outside the platform, which means data and results are no longer in the same place. Researchers currently have to switch manually between OMERO and analysis tools. This switching process involves specific steps: downloading images, preprocessing them, running the model and exporting the results. There is currently no consistent way to run AI-based image analysis directly within OMERO. This has practical implications for researchers. Firstly, switching between different platforms is very time-consuming. Furthermore, manual processing increases the risk of inconsistent results. Finally, it increases the risk of the link between data and metadata getting lost.

1.2 FAIR Data and Reproducibility in Bioimaging

OMERO follows the FAIR principles: data must be Findable, Accessible, Interoperable, and Reusable. FAIR data ensures that data can be shared in a way that promotes its reuse. It is particularly important for bioimaging because microscopy research lags behind other disciplines in terms of data management and sharing, which makes integration difficult[11]. Recent developments have led to many different types of data being combined to gain a more complete picture of what is

happening within the cell. This means that image data no longer stands alone, but must be able to be linked to other types of research data. To this end, it is important that image data is well described and standardised. Image analysis in bioimaging is increasingly being carried out using AI, but these analyses are not always reproducible. Reproducibility is only possible if all steps can be traced. If an analysis is carried out outside OMERO, that traceability cannot be guaranteed. So if an AI analysis is not properly documented, another researcher cannot replicate or verify the result. This poses a problem for the reliability of scientific research. By integrating an AI pipeline into OMERO, the analysis and the data remain in one place, thereby improving traceability. This means that researchers not only know what the outcome is, but also how that outcome was arrived at, which is essential for reproducibility.

1.3 Deep Learning for Automated Image Analysis

Deep learning offers a promising solution for automated image analysis. This form of artificial intelligence can recognise patterns in images that are difficult for the human eye to distinguish. It is trained on large amounts of labelled data and can afterwards analyse new images automatically. This enables tasks such as segmentation, where it is determined for each pixel whether it belongs to a cell; detection, where specific structures are localised; and classification, where images or objects are categorised, to be performed automatically. Deep learning is very powerful, but its application is technically complex. Many biologists do not have the knowledge to train models themselves or write code. For this reason, accessible frameworks have been developed that allow researchers to apply pre-trained models without programming experience. One such framework is BiaPy.

1.3.1 BiaPy

BiaPy is an open-source library and application that streamlines the use of common deep-learning workflows for a wide variety of bioimage analysis tasks [9]. It makes it possible to apply pre-trained models without having to train a model yourself, removing the need to write PyTorch code. Although such tools are becoming increasingly accessible, their integration into existing data platforms such as OMERO is still limited.

1.4 Research Objective

What is currently missing is a reliable way to use AI models with OMERO. This thesis aims to address precisely this issue. A pipeline will be built to connect OMERO with existing AI tools from BiaPy. The focus is on inference only: how existing pre-trained models can be applied directly within OMERO. Training new models falls outside the scope of this work. This pipeline must be modular so that models can be easily swapped. This is necessary because different research groups have different questions. In addition, the pipeline must be reproducible, meaning other researchers must be able to replicate the exact same analysis on their own data; this is important because scientific results must be reproducible. Reproducibility is not only useful, but also essential for scientific verification. Every step in the pipeline must be documented, otherwise, a result cannot be replicated. The pipeline therefore stores not only the results, but also details of which model, version and settings were used. BiaPy will be used as the AI framework, and OMERO's Python API will be used to retrieve images and return results. The FUCCI dataset from Leiden University

is being used as a test case. This dataset is a time-lapse microscopy dataset with 4 biological channels and up to 70 time points per well. The complexity of the dataset makes it representative of the kinds of data researchers typically store in OMERO. In this way, the pipeline makes advanced image analysis accessible to biologists without programming experience.

This leads to the following central research question:

> *How can a modular and reproducible AI inference pipeline be integrated within OMERO for automated image analysis?*

To answer this research question, the following sub-questions will be investigated:

1. What challenges arise when integrating AI-based image analysis tools into OMERO?
2. What design choices are required to create a modular and reproducible AI workflow for microscopic image analysis?

1.5 Thesis overview

The rest of the thesis is structured as follows. Chapter 2 provides background information on the key concepts and technologies used in this thesis, including OMERO, the FAIR data principles, deep learning for bioimaging, BiaPy and StarDist. In addition, existing tools that link OMERO to image analysis are discussed. Chapter 3 describes the subject matter and the implementation of the modular AI pipeline. Chapter 4 presents the results of the pipeline applied to the FUCCI dataset from Leiden University. Finally, Chapter 5 discusses the findings, reflects on the reproducibility and modularity of the pipeline, and provides recommendations for future research.

2 Background and Related Work

2.1 OMERO

OMERO is a data management platform for biological image data. This section describes its architecture, data model and programming interface in more detail.

2.1.1 Architecture

OMERO consists of three components: databases, middleware and clients. The central component is OMERO.server. Figure 1 shows the three-layered architecture of OMERO. It is a Java application that acts as the link between the databases and the users [20]. All requests pass through the server, which provides access to the stored data via a single shared API. This means that clients do not need to communicate directly with the database. Instead, everything is handled via the same interface. Users can connect to the server in various ways. OMERO.web is a browser-based client that allows data to be viewed, annotated and shared. OMERO.insight is a desktop application with similar functionality. In addition, there is a command-line interface for advanced users and scripts. OMERO uses various mechanisms for storage. Raw image data is stored in binary file storage and all the metadata is stored in a relational database, typically PostgreSQL. Large tabular data, such as analysis results, are stored in HDF5 files via OMERO.tables.

Data is imported via Bio-Formats, a library that supports over 100 different microscopy file formats[2]. During import, all metadata is automatically extracted and stored in the database in accordance with the OME data model. This makes OMERO independent of the microscope's file format. Finally, OMERO offers a Scripting Service. This allows Python scripts to be loaded and executed on the server. Scripts can be invoked from a client and then run on the server or on external computing systems.

2.1.2 Data Model

In OMERO, data is organised hierarchically. At the top are Projects, followed by Datasets, and then Images. This arrangement makes it easy to document and access experiments in a clear and organised manner. Additional data can be linked to each image. This can take the form of key-value pairs, tags, annotations or regions of interest (ROIs). ROIs describe specific areas within an image, such as the outline of a cell. This information is stored in the database alongside the image [17]. OMERO uses the OME data model as its basis. This is a standardised way of describing microscopy data. The model defines which metadata belongs to an image and how that metadata is stored. Because this model is open and widely supported, different software tools can read the data in the same way. An important feature of this data model is that it also stores analytical results. Segmentation outputs, calculated features and other calculated results can be linked back to the original image in OMERO. This means that both the raw data and the results of any analysis remain connected in one place [14].

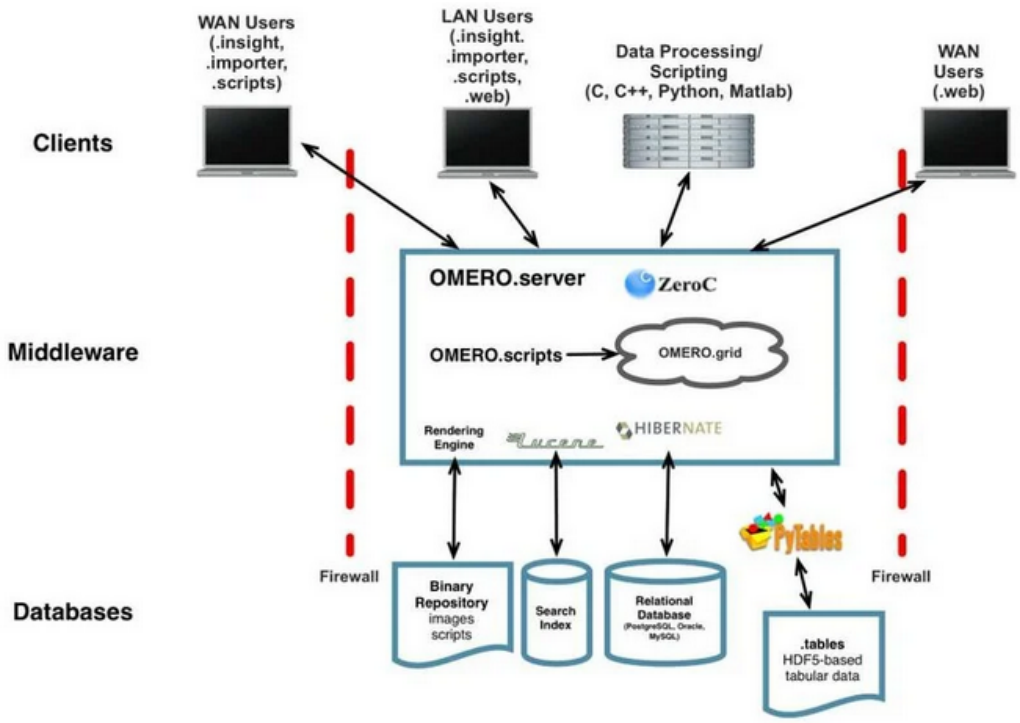

Figure 1: The architecture of OMERO, comprising three layers: clients, middleware and databases. Adapted from [2].

2.1.3 API and Scripting

OMERO has a Python API called omero-py. This API allows developers to control the server via code. This makes it possible to retrieve images, read metadata and send results back to the server. External tools and scripts can thus work directly with data stored in OMERO. The API provides access to all data types within OMERO. These include images, annotations, ROIs and tabular data stored in OMERO.tables. OMERO also has a built-in service for running scripts. Python scripts can be uploaded to the server and executed there. Scripts can be launched from the web interface, making analyses accessible to users without programming experience. The scripts run on the server itself or can be forwarded to external computing systems via OMERO.grid. This makes it possible to perform larger computations on datasets stored in OMERO. Together, the Python API and the Scripting Service make it possible to build workflows that run entirely within OMERO. This is important for the traceability and reproducibility of analyses. However, using this API involves several concrete steps. First, the user must have permission to access the server and must understand how the data is organised. Once the images are retrieved, they must be converted into the format the model expects. Channels relevant to the analysis must be selected, and larger images must be split into tiles before they can be processed by the model. The model output consists of label maps that must be converted back into OMERO objects such as ROIs. Results must be returned as annotations, ROIs or tables in OMERO. All these steps must be logged to ensure the analysis is possible. These two capabilities therefore form the technical basis of the pipeline built in this thesis.

2.1.4 OMERO at Leiden University

Leiden University's central OMERO server contains a wide variety of microscopy data from various research groups [16]. One example is the FUCCI dataset. This is a time-lapse microscopy dataset in which cells are tracked over time using four fluorescent channels: DAPI, FITC, Texas Red and Cy5. Each measurement consists of 70 time points and images of 1024 by 1024 pixels. The dataset provides insight into the cell cycle of individual cells. The size and complexity of this dataset make it a representative example of the type of data stored on the Leiden OMERO server. This is why the FUCCI dataset is used as a test case for the pipeline being built in this thesis.

2.2 FAIR Data and Metadata Standards

Storing data is not enough on its own. For data to be useful to others, it needs to be findable, accessible and well described. This section covers the FAIR principles and the metadata standards that support them in the context of bioimaging.

2.2.1 The FAIR Principles

The FAIR principles were introduced in 2016 by Wilkinson et al. and stand for Findable, Accessible, Interoperable and Reusable. The aim is to manage scientific data in such a way that other researchers can find, understand and reuse it [31]. In the life sciences, these principles are becoming increasingly important as datasets grow larger and more complex. The FAIR principles are particularly relevant to bioimaging. Microscopy data contains not only images but also a great deal of metadata about how the images were acquired [11]. If that metadata is missing or not standardised, it is difficult for other researchers to understand or reuse the data [24]. The FAIR principles are also important for a reproducible AI pipeline. If the input data, the models used and the analysis results are well documented and accessible, another researcher can repeat and verify the analysis. This ties in directly with the objective of this thesis.

2.2.2 Metadata Standards in OMERO

REMBI stands for Recommended Metadata for Biological Images. It is a set of recommendations that describes what metadata should be recorded for a bioimaging experiment. This includes information such as the type of microscope, the dyes used and the settings during imaging. Without this information, it is difficult for other researchers to understand how an experiment was conducted. REMBI was developed as a community initiative and focuses on a wide range of bioimaging applications [24]. The OME data model is the technical foundation on which OMERO is built [10]. This model defines how metadata is stored and exchanged between different tools. Because the model is open and widely supported, tools such as Bio-Formats and OMERO can read and write metadata in a standardised way [2]. The model is regularly updated to support new microscopy techniques and data types. This ensures that OMERO remains relevant for new types of data in the future. In addition to the OME data model, OMERO also supports flexible metadata in the form of key-value pairs and map annotations [17]. This makes it possible to record experimental details that are not included as standard in the OME data model. Examples include cell type, treatment time or the name of a reagent. This flexibility is important because every experiment is different and not all metadata can be standardised in advance. Good metadata is

essential for a reproducible pipeline. If, for every analysis, a record is kept of which images were used, which model was applied and what the settings were, the analysis can be repeated later. OMERO provides the setup for this by linking metadata to the original images and storing analysis results as annotations. This makes it possible to preserve not only the outcome of an analysis, but also the context in which that analysis was carried out.

2.2.3 Reproducibility of AI Analyses

Reproducibility is a major issue in science. Many studies prove difficult to replicate because the methods used are not properly documented [3][15]. This is particularly true for AI analyses. A model may produce good results, but if it is not clear which data was used, which version of the model, and which settings, the result cannot be verified. In deep learning, there are several factors that influence reproducibility. Firstly, the input data is important. If it is not known exactly which images were used for an analysis, another researcher cannot replicate the analysis. Secondly, the model settings play a role. Small differences in parameters can lead to different outcomes. Thirdly, the software version is important. Different versions of a framework such as PyTorch can produce different results on the same data [23]. To make an AI pipeline reproducible, a number of things are required. The input data must be traceable, the models used must be saved, and the settings must be recorded [23]. OMERO provides a good basis for this because data, metadata and analysis results are stored in one place. By building the pipeline as a modular and documented workflow, it becomes possible for other researchers to repeat the same analysis on new data.

2.3 Deep Learning for Bioimaging

The pipeline in this project relies on deep learning models for image analysis. This section explains the key concepts behind these models and how they are used for segmentation. It also covers why pre-trained models are a practical choice for researchers without machine learning expertise.

2.3.1 Convolutional Neural Networks

A CNN stands for Convolutional Neural Network; it is a special type of neural network and it is designed for image processing [13]. A CNN is inspired by the way the human visual system works. It automatically learns to recognise patterns in data, without the need for manual feature extraction. The network consists of several successive layers. These convolutional layers slide small filters, also known as kernels, across the image. These filters are typically 3x3 or 5x5 pixels in size. A convolution operation is a process that calculates the dot product between a filter and a small part of the image. Stride and padding determine how the filter moves across the image and how the edges are handled. Each filter detects a specific pattern, such as a colour, edge or texture. The result of applying a filter is known as a feature map. In such a network, the early layers recognise simple patterns such as edges and corners. The deeper layers combine these into more complex structures such as a cell wall or a nucleus. In between are pooling layers, which reduce the size of the feature maps and make the network more efficient. Ultimately, the fully connected layers at the end make the final prediction.

CNNs work well for images due to their translation invariance. This means that a pattern is recognised regardless of where it is located in the image. Parameters are shared across positions,

which makes the network smaller. Hierarchical learning makes complex patterns understandable to the network. Because CNNs handle images well, they are also used in bioimaging. Microscopy images differ from regular photos; they are often in greyscale, have multiple channels, and sometimes a depth dimension. CNNs can handle this well, including multi-channel data such as the four channels in the FUCCI dataset. They are able to handle noise and variation effectively. Here, they are used for cell segmentation, cell detection or classification. They perform better than traditional image processing methods on complex datasets. These models are used in various bioimaging tools [28], such as BiaPy, CellPose and StarDist.

2.3.2 U-Net for Image Segmentation

An important CNN architecture for image segmentation is U-Net [22]. The name is derived from the architecture's distinctive U-shape. The network was specifically designed to segment cells in microscopy images. The U-Net consists of two parts: an encoder and a decoder. The encoder compresses the image step by step. This is also known as the downsampling path. During this process, the network learns abstract features, but loses track of where those features are located in the image. Spatial information is lost as the resolution decreases. The decoder then reconstructs the segmentation from that compression. This is known as the upsampling path. At the end, the network returns a class label for each pixel. Figure 2 shows this structure clearly. The most

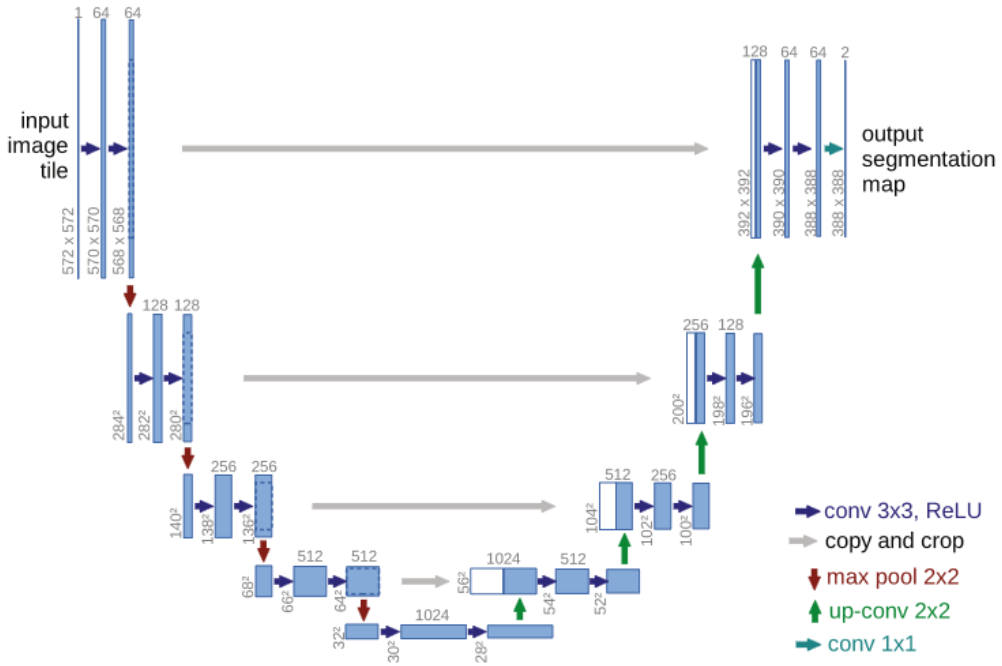

Figure 2: The U-Net architecture, consisting of an encoder (downsampling path, left) and a decoder (upsampling path, right) connected by skip connections. Adapted from [22].

important feature of U-Net is the skip connections. These connect layers from the encoder directly to the corresponding layers in the decoder. As a result, the spatial information lost during compression is nevertheless preserved. This is essential for accurate segmentation at the pixel level.

A major advantage of U-Net is that it performs well with relatively little training data. This is because the network utilises data augmentation during training. This is important in bioimaging, as labelled microscopy data is often scarce. As a result U-Net is very flexible. It works very well with a small amount of labelled data. U-Net is used in many different segmentation tasks. Examples include the segmentation of cell nuclei, cells in fluorescence microscopy and tissues in pathology images. BiaPy uses U-Net inside as one of its architectures. Pre-trained models are already available. This allows researchers to use them immediately, without having to train a network themselves. Another architecture designed specifically for round objects such as cell nuclei is StarDist. Instead of predicting pixel classes directly, StarDist predicts star-shaped polygons around each object. This makes it particularly well-suited to nuclear segmentation in fluorescence microscopy.

2.3.3 Model Inference

Once a CNN has been trained, it can be used for inference. This means that the model is applied to new, unseen data. There is no need to retrain it. The model uses the learned filters and weights to make predictions on new images. Inference is far less demanding than training. During training, gradients must be calculated and weights adjusted. In inference, the image is simply passed through the network. This makes inference suitable for use on standard research hardware [18]. For bioimaging, inference using pre-trained models is particularly valuable. Training a model requires a lot of labelled data and computing time. Many researchers do not have access to this kind of technology. If they use a pre-trained model they can still get the benefits of deep learning [18]. This makes it much easier for them to use AI in their work. Several factors are important in inference. Firstly, the input must match what the model expects. This concerns the format, resolution and number of channels of the image. If the input does not match, the results may be unreliable. Secondly, the settings must be fixed. Small differences in parameters can lead to different outcomes [23]. Thirdly, the version of the model is important. A different model weight yields a different prediction, even on the same data. This makes reproducibility a key consideration in inference. An analysis cannot be repeated if it is not known which model was used and with what settings. By saving the model and configuration alongside the results, the analysis becomes traceable [23]. This is in line with the FAIR principles discussed earlier.

2.3.4 The BioImage Model Zoo

Sharing trained models is important for reproducibility. If a model is not available, others cannot replicate the analysis [23]. The BioImage Model Zoo solves this problem. This is a platform where people can share trained bioimaging models [21]. Each bioimaging model has information about what it was trained for and what kind of data was used to train it. The bioimaging model also says what kind of input it needs. This information helps determine whether a model is suitable for a given analysis. The bioimaging models are all stored in the bioimage.io format. This makes it easy to move bioimaging models between tools. One of the available models is affable-shark. This model is based on U-Net and has been trained for instance segmentation of cell nuclei in 2D microscopy images. It assigns a separate label to each recognised nucleus. In this project, affable-shark is used as the segmentation model. It is directly available via BiaPy and can be applied to new data without modification. As the model is publicly available, other researchers can

retrieve and use exactly the same model. This contributes to the reproducibility of the pipeline.

2.4 BiaPy

BiaPy is the primary AI framework used in this project. This section describes its supported tasks, configuration system and integration with the BioImage Model Zoo in more detail.

2.4.1 Overview and Supported Tasks

BiaPy is an open-source framework that makes deep learning workflows accessible to biomedical researchers. It was developed at the University of the Basque Country and is actively maintained [9]. The aim is to enable complex deep learning tasks to be carried out without the user having to write code. BiaPy forms the primary framework for this pipeline. StarDist is additionally used to demonstrate the modularity of the pipeline. BiaPy supports a wide range of tasks commonly encountered in biomedical image analysis. The most important are segmentation, detection, classification and denoising. In semantic segmentation, each pixel is assigned a class, such as cell or background. In instance segmentation, each individual cell is given a unique label. This distinction is important: to count cells, you need instance segmentation, not semantic segmentation. Detection localises objects in an image without labelling every pixel. Classification assigns a label to a full image or to a detected object. Denoising removes noise from microscopy images, improving the quality of analyses. For this project, instance segmentation is the relevant task. The aim is to segment individual cell nuclei in the FUCCI dataset. BiaPy supports this directly via its instance segmentation workflow. BiaPy is available as a Python library and can also be run via Docker [4]. This does not require a local Python setup. In this project, the Python library is used directly, which allows BiaPy to be called from within the pipeline notebook. BiaPy integrates directly with the BioImage Model Zoo. Pre-trained models can be loaded by specifying a model ID, without any additional setup. This lowers the barrier to entry for researchers who have no experience in training models. In this project, the affable-shark model is used, which is available via the Model Zoo and can be deployed immediately for instance segmentation of cell nuclei. BiaPy is not the only framework for biomedical image analysis. Tools such as StarDist and CellPose are also popular [30][27]. StarDist is used alongside BiaPy as a separate framework to demonstrate the modularity of the pipeline. Unlike affable-shark, the StarDist model does not need to be downloaded via the BioImage Model Zoo. The model has been trained on a wide range of fluorescence microscopy images and works well for cell segmentation without any additional configuration. StarDist and CellPose are compared with BiaPy later in this chapter.

2.4.2 Configuration System

BiaPy uses YAML files for configuration. A YAML file is a text file with a fixed structure which describes all the settings required to perform an analysis. This includes the task type, the input data, the model settings and the output options. By recording everything in a single file, an analysis is fully reproducible after the initial setup of the model [23]. Another researcher can repeat the same analysis by using the same configuration file [4]. A configuration file is made up of several sections [9]. The first section describes the type of task, for example, instance segmentation. The second section describes the input data. This specifies where the images are located, what format they are in and how many channels there are. The third section describes the model. This

specifies which model is used, how many layers it has and what the batch size is. The fourth section describes the output. It specifies where the results are saved and in what format. Listing 1 shows a simplified example of such a configuration file.

Listing 1: A simplified example of a BiaPy YAML configuration file for instance segmentation.

```
PROBLEM:
  TYPE: INSTANCE_SEG
  INSTANCE_SEG:
    DATA_CHANNELS: "BC"

DATA:
  PATCH_SIZE: [256, 256, 1]
  TEST:
    PATH: ../data/processed/
    LOAD_GT: False

MODEL:
  SOURCE: bmz
  BMZ:
    SOURCE_MODEL_ID: "affable-shark"

TEST:
  ENABLE: True
```

For this project, the configuration file has been adapted to the FUCCI dataset. Both the preprocessed images in `data/processed/` and the model output format are defined here. The `DATA_CHANNELS` field specifies the output format of the model. The value `BC` indicates that the model produces a boundary channel and a cell channel, which are combined during post-processing to produce the final instance segmentation mask. The images are 1024 by 1024 pixels, which is too large for the model to handle at once. The `PATCH_SIZE` setting controls how the image is split into smaller patches before inference. For this project, a patch size of 256 by 256 pixels is used, resulting in a grid of 16 patches per 1024 by 1024 image. An important advantage of the YAML system is the connection with the BioImage Model Zoo. A model ID from the Zoo can be specified directly in the configuration file [21]. BiaPy then automatically loads the model. This means that there is no need to manually download or manage model weights. For this project, the affable-shark model is loaded in this way. The configuration file also plays a central role in the reproducibility of the pipeline [23] [31]. The file is stored in OMERO together with the results. This allows anyone to identify the settings used in a given analysis. This is directly in line with the FAIR principles discussed earlier.

2.4.3 Integration with the BioImage Model Zoo

BiaPy is directly integrated with the BioImage Model Zoo. The Model Zoo was previously discussed in section 2.3.4. This section explains how BiaPy uses that integration and what it means for the pipeline. Loading a model from the Model Zoo is done via the configuration file. Instead of specifying a local model file, a model ID is specified. BiaPy then automatically searches

the model in the Model Zoo and downloads the corresponding weights. This means that no model file needs to be managed manually. The model is always loaded in the same version, as long as the same model ID is used [9]. Each model in the Model Zoo has a fixed format. This format is called the bioimage.io format [21]. This format contains not only the model weights, but also metadata. This metadata describes what the model is trained for, what input format it expects and what output it produces. BiaPy automatically reads this metadata. This allows BiaPy to know how to call the model without the user having to set it manually. The affable-shark model is used for this project. This model is trained for instance segmentation of cell nuclei in 2D microscopy images. It is publicly available through the Model Zoo and can be loaded directly through BiaPy. Because the model is publicly available, each researcher can retrieve exactly the same model. This directly contributes to the reproducibility of the pipeline. The integration with the Model Zoo also has practical advantages for the modularity of the pipeline. If another model is desired, only the model ID in the configuration file needs to be modified. This makes the pipeline easily applicable to other datasets and research questions.

2.4.4 Comparison with Similar Tools

BiaPy is not the only framework for automated biomedical image analysis. Tools such as CellPose and StarDist are also widely used within the bioimaging community. In this section, these tools are briefly compared to BiaPy. The focus is on the aspects relevant to this project, how easy it is to configure, Model Zoo integration and accessibility for non-programmers. CellPose is a segmentation model developed for cell segmentation [30]. It works well for a wide variety of cell types and does not require training for new data. CellPose is available as a Python library and as a GUI application. A disadvantage is that CellPose is less flexible in configuration. The settings are given as function arguments in code, not via a configuration file. This makes it more difficult to fully document and reproduce an analysis. In addition, CellPose has no direct integration with the BioImage Model Zoo. StarDist is another popular framework for instance segmentation [27]. It is specifically designed for segmenting round objects such as cell nuclei. StarDist uses a star-convex shape representation, which makes it accurate for this type of object. Like CellPose, StarDist works via Python code and does not have a YAML configuration system. StarDist also has no native integration with the BioImage Model Zoo. BiaPy differs from both tools in a number of ways. It uses YAML configuration files, which makes analyses easier to document and share. It also has direct integration with the BioImage Model Zoo, so models can be loaded via a model ID. StarDist on the other hand is simpler to set up for nucleus segmentation and does not require a configuration file. Both frameworks have their strengths and both are tested in this project. This project tests two independent frameworks for instance segmentation. BiaPy is used with the affable-shark model via the BioImage Model Zoo. StarDist is used directly via its own Python library with the pre-trained `2D_versatile_fluo` model. Unlike affable-shark, this model does not need to be loaded via the BioImage Model Zoo. Switching between the two frameworks requires just a single configuration line in the pipeline. This demonstrates that the pipeline has a modular design. The `2D_versatile_fluo` model has been trained on a wide range of fluorescence microscopy images. This makes the model more robust across different datasets than a model trained on a specific dataset.

2.5 StarDist

StarDist is an open-source framework for instance segmentation of objects in microscopy images [27]. It was developed with a specific purpose in mind, the accurate segmentation of round and oval structures such as cell nuclei. Whilst many segmentation tools are built for general image analysis, StarDist focuses on this specific type of object. This makes it particularly suitable for fluorescence microscopy, because here are all cell nuclei a common object of analysis. StarDist is available as a Python library and can be installed directly via pip. It operates independently of other frameworks such as BiaPy. In this project, StarDist is called directly via its own library, alongside the BiaPy-based approach using affable-shark. This makes it possible to compare both frameworks on the same data.

2.5.1 Star-Convex Polygon Approach

Most segmentation models predict a class for each pixel. StarDist works differently. Instead of pixel classes, it predicts star-shaped polygons around each object. For each possible centre location, the model calculates a set of rays that describe the object's boundary. Together, these rays form a polygon around the centre. This has a practical advantage. Round objects such as cell nuclei fit well into such a star-shaped description. As a result, the boundaries are predicted more accurately than with a pixel-based approach, especially when nuclei are close together. With U-Net, there is a greater risk that adjacent nuclei will merge in the prediction. StarDist avoids this by describing each object as a separate shape.

2.5.2 The 2D_versatile_fluo Model

StarDist offers several pre-trained models. For this project, the `2D_versatile_fluo` model is used. This model has been trained on a large and varied collection of fluorescence microscopy images. The diversity of the training data makes the model robust. It performs well on images that differ in resolution, cell type and dye, without the need for retraining. The model is directly available via the StarDist Python library. No account or external download is required. This allows researchers to start immediately, without additional setup. For the FUCCI dataset, the preprocessed greyscale image produced by maximum projection across all four channels is used as input. The model recognises the nuclei in that channel and assigns a separate label to each nucleus. One difference from affable-shark is that `2D_versatile_fluo` is not loaded via the BioImage Model Zoo. It is built into the StarDist library itself. This simplifies the setup, but also means that the model is less easily replaceable with an alternative from the Model Zoo.

2.6 Related Work

In recent years, various tools have been developed that link OMERO to external analysis tools. This is an active research area, because the need for automated image analysis within OMERO is great [29]. Most existing solutions focus on a specific use or target group. A general, reproducible and modular pipeline is still missing. The existing tools differ greatly in approach. Some tools provide a graphical interface that allows users to start analytics without writing code. Other tools focus on developers and provide a programmable interface. The extent to which reproducibility and modularity are supported also differs per tool. The most relevant tools are discussed in the

following sections. Each tool describes what it does, how it works and where it falls short. The final section of this chapter summarizes all tools and compares them to the pipeline built in this project.

2.6.1 BIOMERO

BIOMERO is a framework that links OMERO to external analysis tools via a workflow engine. The name stands for BioimageMining OMERO. It has been developed to enable researchers to perform analyses on data stored in OMERO. For this they do not have to manually export the data. The data therefore remains on the server during the analysis. BIOMERO uses SLURM, a system for managing calculation tasks on a computer cluster [19] [32]. This makes it possible to process large data sets on external calculation systems. BIOMERO works via a plugin that is installed on the OMERO server. Workflows can be started from the OMERO web interface via this plugin. The workflows are performed on an external SLURM cluster. The results are automatically returned to OMERO. This makes BIOMERO suitable for large-scale analyses on large data sets. Workflow engines are widely used in bioinformatics to make analyses scalable and reproducible [12] [7]. An important advantage of BIOMERO is scalability. Because analyses are performed on an external cluster, the pipeline is not limited by the computing power of the OMERO server. This is relevant for large data sets such as the FUCCI dataset [8]. In addition, BIOMERO is designed to work with existing analysis tools, including tools that run in Docker containers. However, BIOMERO also has limitations that are relevant to this project. First, BIOMERO depends on a SLURM cluster. This means that it is not usable in environments without access to such a cluster. Second, BIOMERO does not offer direct integration with the BioImage Model Zoo. Models must be manually managed. Third, the configuration of workflows in BIOMERO is complex. There is no YAML-based configuration system that fully documents analyses. This makes it more difficult to reproduce and share analyses. BIOMERO is therefore not the most suitable choice for this project. The pipeline built in this project focuses on accessibility and reproducibility without dependence on an external cluster.

2.6.2 MicrobeSEG

MicrobeSEG links instance segmentation to OMERO, the tool is built for segmenting micro-organisms [25]. Users can train and apply models via a graphical interface and the results end up as ROIs in OMERO. An advantage is that images are loaded directly from OMERO. The results go back automatically, where manual export is not necessary. The tool also has a graphical interface, which makes it useful for researchers who are not able to program. However, microbeSEG has limitations. For example, the tool is built for micro-organisms. Therefore, other cell types are not well supported, this makes the tool less suitable for the FUCCI dataset. Integration with the BioImage Model Zoo is also lacking, because models cannot be loaded via a model ID [21]. Further, there is no YAML configuration system, which makes it difficult to document and repeat analyses. In addition, microbeSEG supports both training and inference. But only inference is relevant in this project, the training component therefore adds nothing. It also lacks modularity. The tool is too specific and therefore not suitable for this project.

2.6.3 Other Relevant Tools

In addition to BIOMERO and microbeSEG, there are other tools that link OMERO to analysis tools. These tools are less directly comparable to the pipeline built in this project, but are relevant to discuss. One tool is omero-napari. It is an open-source image viewer plugin for Python that connects OMERO to napari [1]. It allows images stored in OMERO to be opened directly in napari, where further analysis can be carried out. A disadvantage is that omero-napari does not offer an automated pipeline. The user must perform manual steps in napari. This makes the tool less suitable for large-scale analyses and reduces reproducibility. Another tool is omero scripts, it is built into OMERO itself, rather than being a separate tool [2]. It allows Python scripts to be uploaded to the server and executed directly within OMERO. This offers a lot of flexibility, but requires programming experience. In addition, there is no standard configuration system. Each developer implements scripts in their own way. This makes it difficult to share and reproduce scripts. Finally, there is fiji-omero, a plugin that links OMERO to Fiji. Fiji is a widely used image analysis software in biology [26]. With fiji-omero, images from OMERO can be opened in Fiji for further analysis. Like omero-napari, fiji-omero does not offer an automated pipeline. analyses must be performed manually. In addition, Fiji has no direct integration with deep learning frameworks such as BiaPy. Each of these tools addresses part of the problem, but none of the named tools provides a complete solution. The key components: automation, reproducibility, Model Zoo integration and a YAML-based configuration system are never all present in a single tool.

2.6.4 Comparison and Research Gap

The tools discussed show that a lot of work has already been done in the field of OMERO integration. However, each tool lacks one or more properties that are essential for a reproducible and modular pipeline. Table 1 provides an overview of the comparison between the tools discussed and the pipeline built in this project. BIOMERO is closest to tackling this project, as it provides

Tool	YAML	Model Zoo	Reproducible	Modular
BIOMERO	No	No	Partial	Partial
microbeSEG	No	No	Partial	No
omero-napari	No	No	No	No
omero-scripts	No	No	No	Partial
fiji-omero	No	No	No	No
This project	**Yes**	**Yes**	**Yes**	**Yes**

Table 1: Comparison of existing tools with the pipeline developed in this project.

automation and scalability through an external SLURM cluster. However, BIOMERO has important limitations. It requires access to a SLURM cluster, which is not available in every research environment. In addition, a YAML configuration system is missing, making it more difficult to fully document and reproduce analyses. MicrobeSEG offers direct OMERO integration and a graphical interface, but is too specifically designed for micro-organisms. This means that the tool is not suitable for other types of microscopy data, such as the FUCCI dataset used in this project. In addition, both a YAML configuration system and integration with the BioImage Model Zoo are missing. Omero-napari and fiji-omero offer manual workflows suitable for small analyses,

but not for large-scale automation. Both tools lack a configuration system and do not support Model Zoo integration, which limits reproducibility. The central gap is therefore clear. Among the evaluated tools, none of the tools combine OMERO integration with a YAML configuration system, Model Zoo support and full reproducibility in one pipeline. The pipeline built in this project fills exactly this gap. BiaPy is used as an AI framework, the configuration file fully documents each analysis, models are loaded via the Model Zoo and results are returned to OMERO with associated metadata.

3 Design and Implementation

3.1 Software and Hardware Environment

3.1.1 Hardware Specifications

All experiments were done on a Lenovo laptop that runs Windows 11. The laptop has a 13th generation Intel Core i7-13700H processor with 14 cores. It runs at 2.4 GHz. It also has 32 GB of RAM. The laptops graphics are handled by an integrated Intel Iris Xe GPU. There was no GPU available so the inference was done using only the CPU. This means that other researchers can run the pipeline on a standard laptop, without needing a dedicated GPU.

3.1.2 Software Dependencies

The pipeline was developed in Python 3.10.20, managed via a Conda virtual environment. The use of a virtual environment ensures that all package versions are fixed and reproducible. Table 2 lists the main software dependencies.

Package	Version
Python	3.10.20
omero-py	5.22.1
BiaPy	3.6.8
stardist	0.9.2
csbdeep	0.8.2
scipy	1.15.3
matplotlib	3.10.8
numpy	1.26.4
tifffile	2025.5.10
torch	2.3.1+cpu
zeroc-ice	3.6.5

Table 2: Main software dependencies used in this project.

3.1.3 Development Environment

The pipeline was developed in Visual Studio Code using the Conda environment `thesis_env`. Version control was managed via Git. The full list of dependencies can be reproduced using the environment file included in the repository.

3.2 Pipeline Architecture

The pipeline runs from a single Jupyter notebook called `pipeline.ipynb`. Each cell handles one step. The steps run in order, where each cell uses the output of the one before it. This keeps the pipeline easy to follow and straightforward to debug. Figure 3 shows the complete flow of the pipeline. The modular nature of the pipeline lies in the configuration cell. This is the first cell of the notebook and the only part that a researcher needs to adjust. All settings are defined here in

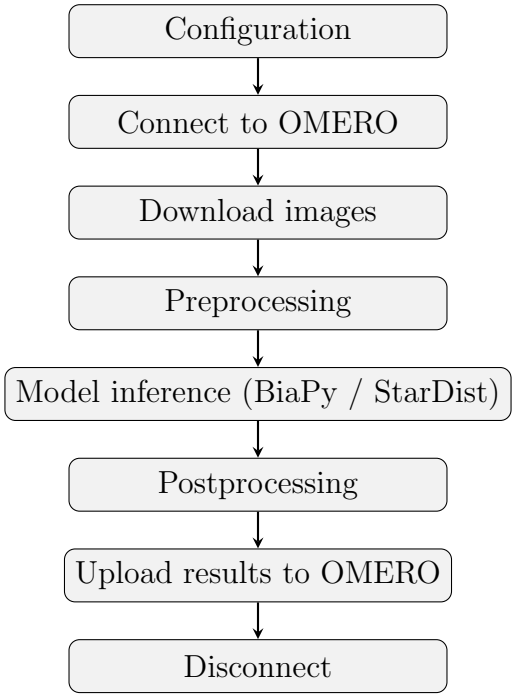

Figure 3: Overview of the pipeline steps.

one place: the plate ID, which wells to process, the timepoint and the model to use. Keeping everything together reduces the chance of mistakes and makes it straightforward to reproduce a run. Switching between models is done via the variable `MODEL_PROFILE_NAME`. A researcher only needs to change this value to use a different model. The corresponding settings are automatically loaded from a profile file named `model_profiles.json`. This file contains the relevant parameters for each model, such as the framework, the model identification, the patch size and the output directory. Listing 2 shows two example profiles.

Listing 2: Two model profiles from model_profiles.json.

```
{
"fucci_affable_shark": {
"task": "instance_segmentation",
"engine": "biapy",
"biapy_config": "models/fucci_segmentation.yaml",
"result_subdir": "masks/fucci_seg",
"run_id": 2,
"patch_size": [256, 256, 1]
},
"fucci_stardist_hoechst": {
"task": "instance_segmentation",
"engine": "stardist",
"pretrained_model": "2D_versatile_fluo",
"input_channel": 0,
"result_subdir": "masks/stardist_hoechst"
}
```

```
}
```

From the selected profile, the pipeline reads the variables `ENGINE`, `TASK` and `CONFIG_FILE`. The value of `ENGINE` tells the inference cell which framework to run. In the inference cell, a choice is made between BiaPy and StarDist based on this value. This makes it possible to switch between two completely independent frameworks with a single configuration rule. Raw images go into `data/raw/`, preprocessed images into `data/processed/` and results into `results/masks/`. Configuration files are kept in `models/` and scripts in `scripts/`. Keeping data, code and configuration in separate folders makes the project easier to navigate and less prone to errors.

3.3 OMERO Connection and Data Retrieval

The pipeline uses the `omero-py` library to connect to OMERO. The connection runs through a `BlitzGateway` client that starts once when the notebook opens. All cells after that share the same connection. When the notebook finishes, the connection closes. Credentials are never written into the code. At startup, the notebook asks for a username and password. The username is entered via `input()` and the password via `getpass()`, which keeps it hidden while typing. Each researcher logs in with their own account, so no shared passwords end up in the repository. To prevent the connection from timing out during long inference runs, a background thread is started after connecting to OMERO. The thread sends a keepalive ping to the OMERO server every five minutes using `conn.keepAlive()`. This thread runs as a daemon process that terminates automatically when the notebook kernel shuts down. The FUCCI dataset is stored as a high-throughput plate in OMERO, with the hierarchy `Plate → Well → WellSample → Image`. The pipeline navigates this structure based on the settings in the configuration cell. A researcher specifies the plate ID, which wells to analyse and which timepoint to use. It is also possible to limit the download to one field per well instead of all four. The pipeline then filters the wells by position and retrieves the corresponding images. Each image has four channels, one timepoint and one Z-plane. The raw data is stored as a NumPy array and saved as a TIFF file to `data/raw/`.

3.4 Preprocessing

Once the images have been downloaded from OMERO, they are prepared for the model. The raw images have four channels: DAPI, FITC, Texas Red and Cy5. Each channel contains biological information about a different aspect of the cell. However, most segmentation models require a single input channel. The four channels are therefore merged into a single greyscale image. This merging is done via a maximum projection across the channel dimension. For each pixel, the highest value across all channels is taken. This ensures that the nuclear signal from all channels is retained in the final image. Both BiaPy and StarDist receive this merged greyscale image as input. The maximum projection is applied regardless of which model is used. An alternative would be to use only the DAPI channel, but the maximum projection gives a more robust result because it is less dependent on the quality of a single channel. After merging, the image is normalised. The pixel values are converted to a range of 0 to 1. This is because a 16-bit image can have pixel values ranging from 0 to 65,535, which is a much larger range than the 0 to 1 that models expect. Models are sensitive to the scale of the input. Normalisation eliminates large differences in intensity between images, which makes the predictions more stable. The pre-processed images are saved as

TIFF files in `data/processed/`. From there, they are read by the model. By saving the pre-processed images separately, it is possible to check the preprocessing step independently before starting the inference. Tiling is not performed manually. The 1024 by 1024 pixel images are too large to feed into most models at once. BiaPy handles this by splitting them internally into patches of 256 by 256 pixels. This results in a grid of 16 patches per image. After inference, the patches are automatically merged back into a complete mask. The StarDist model works differently; this model processes the entire image in one go and does not require tiling. StarDist processed the full image at once because it predicts polygon shapes rather than dense pixel maps. It therefore requires less memory than a full pixel-level prediction.

3.5 Model Inference

The inference step is at the heart of the pipeline. This is where the pre-processed images are fed through the segmentation model. The `ENGINE` variable in the profile determines which framework runs. Setting it to `biapy` starts BiaPy, setting it to `stardist` starts StarDist. With BiaPy, inference is performed via a YAML configuration file. This file describes the task, the input data, the model settings and the output location. The `affable-shark` model is automatically loaded from the local cache via the BioImage Model Zoo integration. BiaPy divides the images internally into patches, performs the inference, and then merges the patches back together. The output consists of boundary maps and foreground probability maps. Watershed post-processing is then applied to separate individual nuclei from one another. Each nucleus is assigned a unique integer label in the final mask. With StarDist, the inference process is different. The `2D_versatile_fluo` model is loaded directly via the StarDist Python library. The image is normalised based on the 1st and 99.8th percentile values. The model predicts a star-shaped polygon around each nucleus. Where predictions overlap, Non-Maximum Suppression removes the weaker ones. What remains is a 2D mask where every nucleus carries a unique label. Both models therefore produce an instance segmentation mask as output. The mask is saved as a TIFF file in the output directory specified in the model profile. From there, it is picked up by the post-processing step.

3.6 Postprocessing and Result Upload

After inference, the segmentation masks are processed in the postprocessing step. For both models, the raw output is an instance mask where each nucleus carries a unique integer label. This mask is converted to a binary representation for uploading to OMERO. For BiaPy, the BMZ cell channel is saved and uploaded rather than a simple binary mask. This is the model's raw foreground probability output, showing nucleus regions as white outlines on a black background. The nucleus count is derived separately from the postprocessed instance mask, with a minimum area filter of 50 pixels applied to remove small watershed artefacts. This threshold was selected empirically based on visual inspection of the resulting masks. For StarDist, the instance mask is converted directly to a binary mask where nucleus pixels are set to 255 and the background pixels to 0. For BiaPy the background label is 1 instead of 0, so a threshold of greater than 1 is applied. For StarDist the threshold of greater than 0 is used. The nucleus count for each image is saved to a JSON file during postprocessing. This is necessary because for BiaPy the uploaded image is the BMZ cell channel, not the instance mask itself. Without this intermediate file, the upload step would have to recalculate the count from the cell channel image. That image does not contain individual nucleus

labels and would therefore produce an incorrect result. This JSON file is read by the upload step to ensure the count stored in OMERO matches the filtered instance mask rather than the uploaded visualisation image.

After post-processing, the masks are uploaded to OMERO. The pipeline automatically creates a project and dataset in OMERO for this purpose, based on the name of the model profile. The masks are saved as new image objects in that dataset. This means that the results are immediately visible in the OMERO.web interface, alongside the original images. Metadata is added to each uploaded mask in the form of key-value pairs. Clicking on an image in OMERO.web shows these values directly. They include the number of segmented nuclei, the well position, plate ID, timepoint, task, model name, model profile and the date the analysis was run. This makes the results immediately traceable without the need for external log files.

3.7 Reproducibility Measures

Reproducibility is a key component of the pipeline. An analysis must be able to be repeated by another researcher with exactly the same result. For this reason, various measures have been put in place. The most important component is the configuration cell. All parameters are stored in one place. Anyone who enters the same values will obtain the same analysis. This applies to the plate ID, the wells, the time and the model profile. The model profiles are stored in `model_profiles.json`. This file is version-controlled via Git. This ensures that it is always possible to trace which model was used with which settings. BiaPy also automatically saves a copy of the YAML configuration file alongside the results. All results in OMERO are accompanied by metadata. The model name, profile, date and other parameters are stored as key-value pairs for each mask. This makes the results traceable without the need for external log files. The nucleus count for each processed image is saved to a JSON file alongside the binary masks. It ensures that the count that is stored in OMERO as a key-value pair. It always reflects the filtered instance mask, regardless of what image is uploaded for visualisation. The full pipeline, including the notebook, model profiles, environment file and configuration files, is available in the project repository.

4 Results

4.1 Pipeline Execution

The entire pipeline has been successfully executed for both models. After changing one line in the first cell, the pipeline takes over. It downloads images from OMERO, preprocesses them, runs inference, processes the masks and uploads the results back. The only manual step is logging in at the start. Table 3 shows how long each step takes per image for both models, measured on a CPU without a GPU.

Step	BiaPy	StarDist
Download	∼2 sec	∼2 sec
Preprocessing	∼0.1 sec	∼0.1 sec
Inference	∼17 sec	∼170 sec
Postprocessing	∼0.5 sec	∼0.5 sec
Upload to OMERO	∼0.5 sec	∼0.5 sec
Total per image	**∼1 min**	**∼3 min**
Total 16 images	**∼5.5 min**	**∼48 min**

Table 3: Average runtimes per step for both models.

A number of technical issues arose during development. The model weights for affable-shark could not be downloaded automatically due to timeouts on the bioimage.io server. This was resolved by manually downloading the files and placing them in the local BiaPy cache. Additionally, BiaPy and StarDist conflict when running in the same Python session. A kernel restart is required when switching models. Finally, the `zeroc-ice` dependency was not compatible with Python 3.11 on Windows. This was resolved by using Python 3.10 with a pre-built wheel from Glencoe Software.

Processing the first image always takes longer than the ones that follow. The model has to start up, weights are read from disk and PyTorch runs certain operations for the first time. From the second image onwards, all of this is already done. The times in Table 3 are averages across 16 images and are therefore lower than what a single isolated run would show.

4.2 Segmentation Results - BiaPy (affable-shark)

The number of detected regions varied considerably per field. Table 4 shows the counts per well and field after applying a minimum area filter of 50 pixels. For BiaPy, the BMZ cell channel is uploaded to OMERO rather than a simple binary mask. This image shows the model's foreground probability output as white nucleus outlines on a black background, which gives a cleaner visual result than the raw instance mask. The maximum projection during preprocessing produced clean greyscale images that the model could process without difficulty. The semantic segmentation, determining where nuclei are located, worked well.

However, the instance segmentation was less reliable. The watershed postprocessing, which is responsible for separating individual nuclei, merged groups of three to five adjacent nuclei into a single large region. This resulted in large patches in the mask rather than distinct nuclear contours.

This may be due to a mismatch between the training data and the FUCCI dataset. Affable-shark was primarily trained on well-separated, DAPI-stained nuclei. The FUCCI dataset contains densely packed nuclei with irregular morphology due to the fluorescence labelling of the cell cycle. The model recognises where nuclei are, but fails to correctly determine the boundaries between adjacent ones. This makes the nucleus counts less reliable for quantitative analyses. Well C02 Field 3 was excluded from the average nuclei count because the image quality was insufficient for reliable segmentation. The image shows significant artefacts in the upper left and lower left corners, where a bright blob is visible that do not corresponds to cell nuclei. This is likely caused by out-of-focus material or debris on the well surface. The other 15 fields did not show comparable artefacts and were used for the reported averages.

Well	**Field 1**	**Field 2**	**Field 3**	**Field 4**
B02	243	281	262	289
B03	226	267	313	292
C02	253	261	-	297
C03	236	252	226	264
Average		∼264		

Table 4: Nucleus counts per field for BiaPy (affable-shark) after applying a 50 pixel minimum area filter.

4.3 Segmentation Results - StarDist (`2D_versatile_fluo`)

The StarDist model detected an average of 258 nuclei per image. Table 5 shows the counts per well and field. Non-Maximum Suppression in StarDist removes overlapping predictions, whereas BiaPy merges touching nuclei via watershed into larger regions. The visual quality of the masks was clearly better than that of BiaPy. Each nucleus was assigned a tight polygon boundary as a unique instance label. The masks showed well-separated nuclear shapes with clear contours that corresponded to the morphology in the original fluorescence images. This makes StarDist's nucleus counts more reliable for quantitative analyses. However, a few false-positive detections were observed. Small artefacts in the background were occasionally detected as nuclei. This was most pronounced at the edges of the wells, where cell density and illumination varied. The default threshold of 0.5 was used, as it produced visually acceptable results without further optimisation. It reduced the number of false positives, but did not resolve the issue completely. Well C02 Field 3 was excluded from the StarDist results for the same reason as described for BiaPy.

Well	**Field 1**	**Field 2**	**Field 3**	**Field 4**
B02	233	229	267	257
B03	243	264	291	304
C02	251	271	-	283
C03	245	253	230	249
Average		∼258		

Table 5: Nucleus counts per field for StarDist (2D_versatile_fluo).

4.4 Comparison between models

Both models produce an instance segmentation mask, but the quality differs significantly. StarDist performs better on the FUCCI dataset when it comes to separating individual instances. BiaPy does recognise where instances are located, but fails to separate adjacent instances from one another. The key differences between the two models are summarised in Table 6. Switching

Aspect	BiaPy (affable-shark)	StarDist (2D_versatile_fluo)
Average nuclei count	∼264	∼258
Instance separation	Poor	Good
False positives	Low	Moderate
Inference time per image	∼17 sec	∼170 sec
Model source	BioImage Model Zoo	Built-in StarDist library
Suitable for quantification	No	Yes
Lines to switch model	1	1

Table 6: Comparison of BiaPy and StarDist on the FUCCI dataset. Switching is configuration-driven, although environment conflicts currently require kernel restarts.

between the two models came down to changing one line in the configuration cell. The rest of the pipeline adjusted automatically. This demonstrates that the modular design works as intended.

4.5 OMERO integration

The masks for both models have been successfully uploaded to OMERO. They have been saved as new image objects under the `Segmentation_Results` project, in model-specific datasets:

- `fucci_T0_fucci_affable_shark` — BiaPy results (BMZ cell channel)
- `fucci_T0_fucci_stardist_hoechst` — StarDist results (binary mask)

For BiaPy, the uploaded image shows the BMZ cell channel with nucleus outlines visible as white contours on a black background. For StarDist, a binary mask is uploaded showing white filled nuclei on a black background. The results are immediately visible in OMERO.web, alongside the original images. Key-value pairs are linked to each uploaded mask. The right panel of OMERO.web shows these values directly when clicking on an image. The metadata makes the results directly traceable within OMERO. A researcher can see which model was used, on what date and with what settings, without consulting external log files.

4.6 Figures and Visualisations

Figure 4 shows the greyscale image of Well B02 Field 1 following the preprocessing step. The maximum intensity projection across all four channels clearly reveals the nuclei as bright structures against a dark background. The nuclei vary in brightness, which is biologically expected for a population of cells in different cell cycle phases. The background is virtually black, indicating that the normalisation has worked correctly.

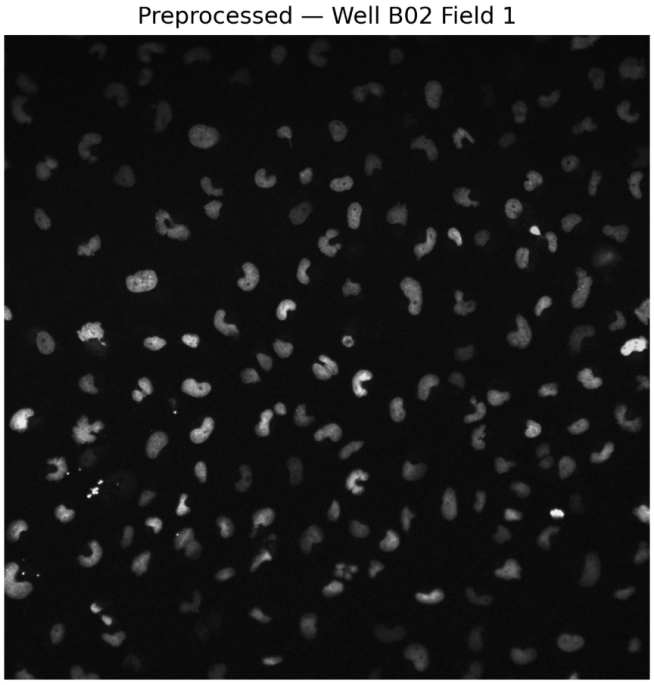

Figure 4: Preprocessed greyscale image of Well B02 Field 1, produced by maximum intensity projection across all four fluorescence channels.

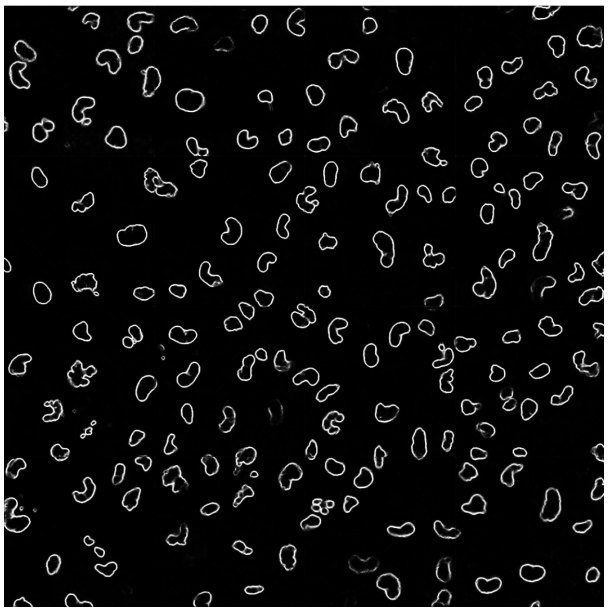

Figure 5: BMZ cell channel output of the affable-shark model for Well B02 Field 1. Each white outline represents a detected nucleus region. The outlines correspond to the model's foreground probability output after watershed post-processing. Some outlines enclose multiple adjacent nuclei, reflecting the instance separation limitations of the watershed algorithm on densely packed FUCCI data.

Figure 5 shows the BMZ cell channel of the affable-shark model alongside the preprocessed input image. The general nuclei detection is correct, the model does recognise where the nuclei are located. However, there are clearly large white regions visible that cover several nuclei from the original image. The watershed post-processing merged touching nuclei into single regions instead of separating them.

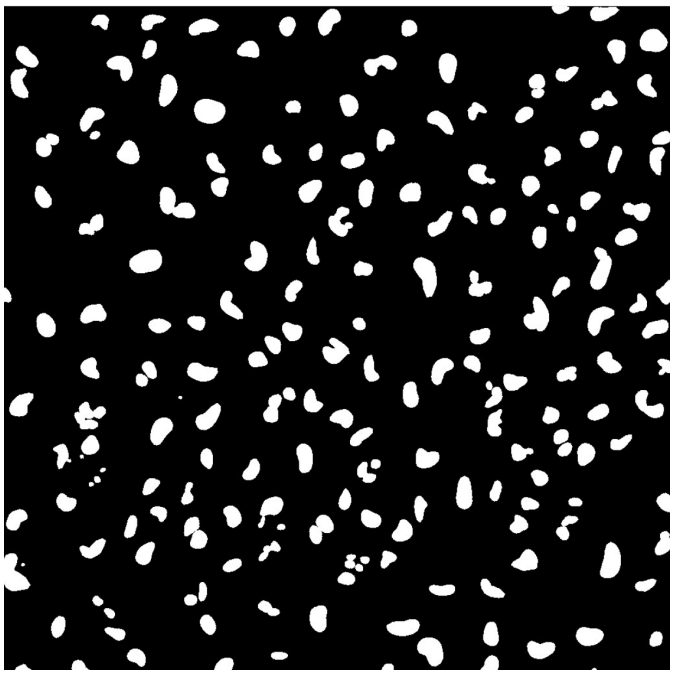

Figure 6: Binary segmentation mask produced by StarDist (2D_versatile_fluo) for Well B02 Field 1. Each white region represents a single segmented nucleus. Compared to the affable-shark output, the nuclei are better separated from one another, with clear gaps between adjacent cells. A small number of false-positive detections are visible as tiny white dots, particularly in denser regions of the image.

Figure 6 shows what the same image looks like with the StarDist mask. The individual nucleus shapes are clearly distinct from one another. The contours correspond well with the morphology in the original image. Oval, crescent-shaped and round nuclei are all correctly recognised. Compared with the BiaPy mask, there is much less merging of adjacent nuclei.

Figure 7 shows a screenshot of OMERO.web displaying the key-value pairs for a single uploaded mask. The `nuclei_count` field shows how many nuclei were detected. The `model` and `model_profile` fields show which model was used. The `run_date` records when the analysis ran. All of this is visible without opening the image. This makes every result immediately traceable within OMERO.

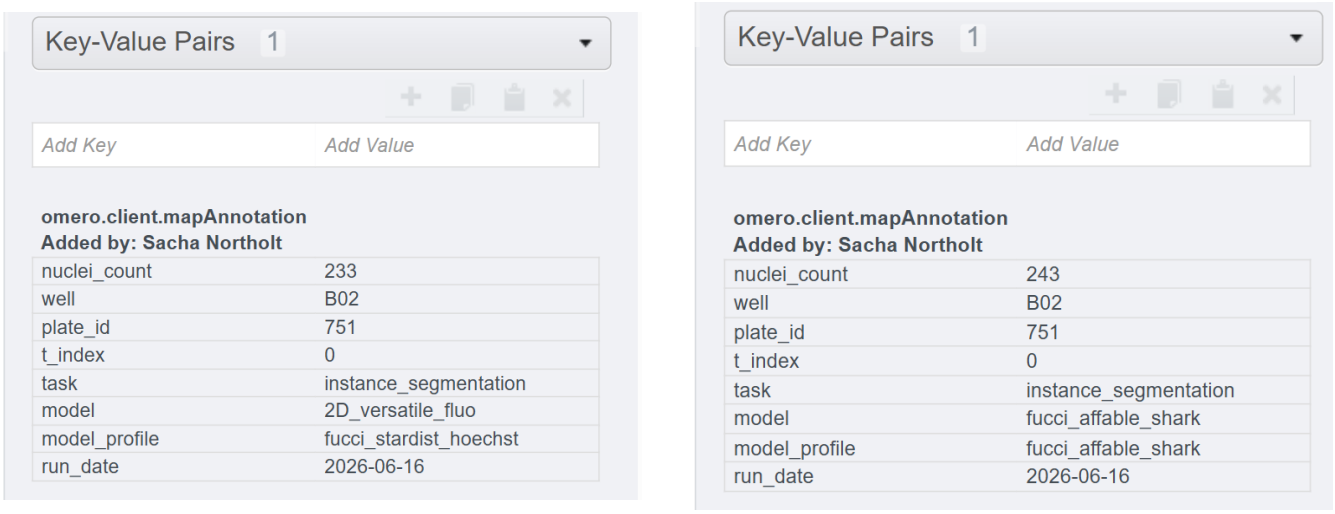

Figure 7: Key-value pairs stored in OMERO.web for an uploaded StarDist mask (left) and BiaPy mask (right). Each mask is annotated with the nuclei count, well position, plate ID, timepoint, task, model name, model profile and run date.

5 Discussion and Conclusion

5.1 Discussion

The aim of this research was to build a modular and reproducible AI pipeline within OMERO. The pipeline operates end-to-end and supports two frameworks. However, there are also aspects that did not go so well during this project or could be improved. The modularity works as intended, because switching between BiaPy and StarDist requires just one line of code and the rest adapts on its own. This shows that the architecture around model profiles was chosen well. In theory, any AI model can be added to the pipeline. Only a Python interface is needed, as well as an output format that matches what the post-processing step expects.

The second sub-question of this thesis asked what design choices are required to create a modular and repeatable workflow. All settings are defined in a single configuration cell, which keeps the pipeline simple and reduces the risk of mistakes. The model settings are in a separate profile file. Therefore when a model is swapped, it does not touch anything else in the code. Finally, metadata is written back to OMERO after each run. These three choices together make the pipeline modular and reproducible.

Reproducibility is ensured in several ways. All settings are located in a single place within the configuration cell. Model profiles are stored under version control. Metadata is automatically written back to OMERO. This makes every analysis traceable without the need for external log files.

All of this fits well with the FAIR principles. The F for Findable is supported by the fact that every result is stored in OMERO with structured metadata, including model name, date and well position. The A for Accessible means that the results are immediately available via OMERO.web without a researcher having to rerun the pipeline. The I for Interoperable is supported by the fact that the pipeline works with open standards such as TIFF and the OME data model. The R in Reusable is ensured by the fact that the configuration files and model profiles are stored in the repository, so that another researcher can repeat exactly the same analysis on new data. It should be noted, however, that the model's warm-up overhead affects execution times. The reported times are averages across 16 images and are therefore lower than what a single isolated run would take.

The segmentation results were mixed, surprisingly, StarDist performed much better on the FUCCI dataset than BiaPy with affable-shark, despite being simpler to set up. Affable-shark was trained on well-separated nuclei, while the FUCCI dataset has densely packed ones, that mismatch is likely the cause. The watershed step needs clear boundaries to separate nuclei. When those boundaries are weak, it floods across them and merges several nuclei into one blob. This is a known limitation of watershed on dense data and it is not specific to BiaPy. StarDist solves this by predicting polygons around each nucleus directly, that makes it more robust here. The modular pipeline makes it easy to swap in a better model when results fall short.

Several technical issues arose during development.The kernel conflict between BiaPy and StarDist was the most difficult to resolve. A restart is needed every time models are switched, which breaks the flow. The affable-shark weights also had to be downloaded by hand due to server timeouts.Resolving these issues required considerable trial and error. These challenges are directly

relevant to the first sub-question. The main obstacles were version conflicts, network issues and format mismatches, each of which required a separate solution.

Moving on to compare BIOMERO and microbeSEG, this pipeline has clear advantages. The YAML setup makes analyses very easy to document and the Model Zoo connection makes models easy to swap. However, MicrobeSEG only works for micro-organisms and it has no setup file system. This is what makes it hard to repeat analyses. Therefore, this pipeline makes it rather general. BIOMERO scales better on large datasets via a cluster, which this pipeline cannot match. For this project that was fine, but with hundreds of wells it would become a bottleneck.

This pipeline also has limitations. It was tested on one dataset, one time point and a limited number of wells. How it performs on other data is unknown and the post-processing is tuned to the FUCCI dataset. So, other datasets may need different post-processing. All experiments ran without a GPU, so it is hard to say exactly how fast it runs with a GPU.

5.2 Further Research

The pipeline provides a functional foundation; however, there is room for improvement.

First of all, GPU support is a clear next step. StarDist alone took 48 minutes for 16 images on CPU, so a full plate with 70 time points would take very long. And PyTorch supports CUDA, so adding GPU acceleration would not require that many changes to the pipeline.

Second, the pipeline only handles one time point per run now. For a full time-lapse analysis of the FUCCI dataset, it needs to loop over all 70 time points automatically. Combined with GPU support, a full plate could run in one go, which would be more time efficient.

The session conflict between BiaPy and StarDist remains a challenge. Running each model in its own Docker container would resolve this. Then the pipeline would call the right container based on the model profile and new models could be added without touching the existing setup.

On top of that, affable-shark did not perform optimally, so CellPose would be worth trying next. It performs well across a wider variety of cell types. Or fine-tuning a StarDist model on FUCCI images is another option. The modular design makes either approach straightforward to add.

Additionally, the pipeline runs on a laptop, which is not scalable. ALICE, the HPC cluster at Leiden University, would allow much larger datasets to be processed. The inference step would be done at ALICE as a separate job to enhance the speed of the pipeline.

Finally, BiaPy supports more than segmentation. It supports detection, classification and denoising as well. Extending the pipeline to support these tasks via the same setup system would make it useful to a wider group of researchers. For FUCCI specifically, because classifying cells by cycle phase based on channel intensity is a natural next step. A first attempt was made as a proof of concept, but it needs more work.

5.3 Conclusion

This research shows that a modular and repeatable AI pipeline within OMERO has been created. Images are retrieved from OMERO, processed by the model, and the results are written back,

including metadata. In the pipeline, there are two models that work and it requires only one line to switch between them. This answers the research question. Reaching this point required significant effort. Python version conflicts, manual downloads and session conflicts all had to be resolved along the way.

The pipeline is modular and repeatable because of three choices. First, all settings are centralised in a single configuration cell. Secondly, the model profiles are in a separate file. Finally, the metadata is written back to OMERO after every run.

StarDist outperformed BiaPy on this dataset. This was not expected. It shows that the model matters more than the pipeline around it, so the model is what determines the quality of the output.

The pipeline makes AI analysis in OMERO accessible without programming knowledge, this serves as a foundation for future work. GPU support and multi-timepoint processing are the obvious next steps. Once that is in place, it could become a practical tool for the bioimaging community.

References

[1] J. Ahlers, D. Althviz Moré, O. Amsalem, A. Anderson, G. Bokota, P. Boone, J. Bragantini, G. Buckley, A. Burt, M. Bussonnier, et al. napari: a multi-dimensional image viewer for Python, 2023. Version 0.4.18.

[2] C. Allan, J.M. Burel, J. Moore, C. Blackburn, M. Linkert, S. Loynton, D. MacDonald, W.J. Moore, C. Neves, A. Patterson, M. Porter, A. Tarkowska, B. Loranger, J. Avondo, I. Lagerstedt, L. Lianas, S. Leo, K. Hands, R.T. Hay, A. Patwardhan, C. Best, G.J. Kleywegt, G. Zanetti, and J.R. Swedlow. OMERO: flexible, model-driven data management for experimental biology. *Nature Methods*, 9:245–253, 2012.

[3] M. Baker. 1,500 scientists lift the lid on reproducibility. *Nature*, 533:452–454, 2016.

[4] C. Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, 2015.

[5] J.M. Burel, S. Besson, C. Blackburn, M. Carroll, R.K. Ferguson, H. Flynn, K. Gillen, R. Leigh, S. Li, D. Lindner, M. Linkert, W.J. Moore, B. Ramalingam, E. Rozbicki, A. Tarkowska, P. Walczysko, C. Allan, J. Moore, and J.R. Swedlow. Publishing and sharing multi-dimensional image data with OMERO. *Mammalian Genome*, 26:441–447, 2015.

[6] I. Cunha, E. Latron, S. Bauer, D. Sage, and J. Griffíe. Machine learning in microscopy – insights, opportunities and challenges. *Journal of Cell Science*, 137(20):jcs262095, 2024.

[7] P. Di Tommaso, M. Chatzou, E.W. Floden, P.P. Barja, E. Palumbo, and C. Notredame. Nextflow enables reproducible computational workflows. *Nature Biotechnology*, 35:316–319, 2017.

[8] K.W. Eliceiri, M.R. Berthold, I.G. Goldberg, L. Ibáñez, B.S. Manjunath, M.E. Martone, R.F. Murphy, H. Peng, A.L. Plant, B. Roysam, N. Stuurman, J.R. Swedlow, P. Tomancak, and A.E. Carpenter. Biological imaging software tools. *Nature Methods*, 9:697–710, 2012.

[9] D. Franco-Barranco, J.A. Andrés-San Román, I. Hidalgo-Cenalmor, L. Backová, A. González-Marfil, C. Caporal, A. Chessel, P. Gómez-Gálvez, L.M. Escudero, D. Wei, A. Muñoz-Barrutia, and I. Arganda-Carreras. BiaPy: accessible deep learning on bioimages. *Nature Methods*, 22:1124–1126, 2025.

[10] I.G. Goldberg, C. Allan, J.M. Burel, D. Creager, A. Falconi, H. Hochheiser, J. Johnston, J. Mellen, P.K. Sorger, and J.R. Swedlow. The open microscopy environment (OME) data model and XML file: open tools for informatics and quantitative analysis in biological imaging. *Genome Biology*, 6:R47, 2005.

[11] R. Hosseini, M. Vlasveld, J. Willemse, B. van de Water, S.E. Le Dévédec, and K.J. Wolstencroft. FAIR high content screening in bioimaging. *Scientific Data*, 10:462, 2023.

[12] J. Köster and S. Rahmann. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19):2520–2522, 2012.

[13] A. Krizhevsky, I. Sutskever, and G.E. Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 25, pages 1106–1114, 2012.

[14] L. Kuhn Cuellar, A. Friedrich, G. Gabernet, L. de la Garza, S. Fillinger, A. Seyboldt, T. Koch, S. zur Oven-Krockhaus, F. Wanke, S. Richter, W.M. Thaiss, M. Horger, N. Malek, K. Harter, M. Bitzer, and S. Nahnsen. A data management infrastructure for the integration of imaging and omics data in life sciences. *BMC Bioinformatics*, 23:61, 2022.

[15] R.F. Laine, I. Arganda-Carreras, R. Henriques, and G. Jacquemet. Avoiding a replication crisis in deep-learning-based bioimage analysis. *Nature Methods*, 18:1136–1144, 2021.

[16] Leiden University, Cell Observatory. OMERO — Leiden University, 2020. Accessed: May 27, 2026.

[17] S. Li, S. Besson, C. Blackburn, M. Carroll, R.K. Ferguson, H. Flynn, K. Gillen, R. Leigh, D. Lindner, M. Linkert, W.J. Moore, B. Ramalingam, E. Rozbicki, G. Rustici, A. Tarkowska, P. Walczysko, E. Williams, C. Allan, J.M. Burel, J. Moore, and J.R. Swedlow. Metadata management for high content screening in OMERO. *Methods*, 96:27–32, 2016.

[18] G. Litjens, T. Kooi, B.E. Bejnordi, A.A.A. Setio, F. Ciompi, M. Ghafoorian, J.A.W.M. van der Laak, B. van Ginneken, and C.I. Sánchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60–88, 2017.

[19] T.T. Luik, R. Rosas-Bertolini, E.A.J. Reits, R.A. Hoebe, and P.M. Krawczyk. BIOMERO: A scalable and extensible image analysis framework. *Patterns*, 5(8):101024, 2024.

[20] J. Moore, M. Linkert, C. Blackburn, M. Carroll, R.K. Ferguson, H. Flynn, K. Gillen, R. Leigh, S. Li, D. Lindner, W.J. Moore, A.J. Patterson, B. Pindelski, B. Ramalingam, E. Rozbicki, A. Tarkowska, P. Walczysko, C. Allan, J.M. Burel, and J.R. Swedlow. OMERO and Bio-Formats 5: flexible access to large bioimaging datasets at scale. In *Proceedings of SPIE*, volume 9413, 2015.

[21] W. Ouyang, F. Beuttenmueller, E. Gómez de Mariscal, C. Pape, T. Burke, C. Garcia-López de Haro, C. Russell, L. Moya-Sans, C. de-la Torre-Gutiérrez, D. Schmidt, D. Kutra, M. Novikov, M. Weigert, U. Schmidt, P. Bankhead, G. Jacquemet, D. Sage, R. Henriques, A. Muñoz-Barrutia, E. Lundberg, F. Jug, and A. Kreshuk. BioImage Model Zoo: A community-driven resource for accessible deep learning in BioImage analysis, 2022.

[22] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, 2015.

[23] G.K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig. Ten simple rules for reproducible computational research. *PLoS Computational Biology*, 9(10):e1003285, 2013.

[24] U. Sarkans, W. Chiu, L. Collinson, M.C. Darrow, J. Ellenberg, D. Grunwald, J.K. Hériché, A. Iudin, G.G. Martins, T. Meehan, K. Narayan, A. Patwardhan, M.R.G. Russell, H.R. Saibil, C. Strambio-De-Castillia, J.R. Swedlow, C. Tischer, V. Uhlmann, P. Verkade, M. Barlow, O. Bayraktar, E. Birney, C. Catavitello, C. Cawthorne, and A. Brazma. REMBI:

Recommended metadata for biological images—enabling reuse of microscopy data in biology. *Nature Methods*, 18:1418–1422, 2021.

[25] T. Scherr, J. Seiffarth, B. Wollenhaupt, O. Neumann, M.P. Schilling, D. Kohlheyer, H. Scharr, K. Nöh, and R. Mikut. microbeseg: A deep learning software tool with OMERO data management for efficient and accurate cell segmentation. *PLOS ONE*, 2022.

[26] J. Schindelin, C.T. Rueden, M.C. Hiner, and K.W. Eliceiri. The ImageJ ecosystem: An open platform for biomedical image analysis. *Molecular Reproduction and Development*, 82(7-8):518–529, 2015.

[27] U. Schmidt, M. Weigert, C. Broaddus, and G. Myers. Cell detection with star-convex polygons. In *Medical Image Computing and Computer Assisted Intervention – MICCAI 2018*, pages 265–273, 2018.

[28] D. Shen, G. Wu, and H.I. Suk. Deep learning in medical image analysis. *Annual Review of Biomedical Engineering*, 19:221–248, 2017.

[29] D.R. Stirling, M.J. Swain-Bowden, A.M. Lucas, A.E. Carpenter, B.A. Cimini, and A. Goodman. CellProfiler 4: improvements in speed, utility and usability. *BMC Bioinformatics*, 22:433, 2021.

[30] C. Stringer, T. Wang, M. Michaelos, and M. Pachitariu. Cellpose: a generalist algorithm for cellular segmentation. *Nature Methods*, 18:100–106, 2021.

[31] M.D. Wilkinson, M. Dumontier, I.J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.W. Boiten, L.B. da Silva Santos, P.E. Bourne, J. Bouwman, A.J. Brookes, T. Clark, M. Crosas, I. Dillo, O. Dumon, S. Edmunds, C.T. Evelo, R. Finkers, A. Gonzalez-Beltran, A.J.G. Gray, P. Groth, C. Goble, J.S. Grethe, and B. Mons. The FAIR guiding principles for scientific data management and stewardship. *Scientific Data*, 3:160018, 2016.

[32] A.B. Yoo, M.A. Jette, and M. Grondona. SLURM: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing – JSSPP 2003*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60, 2003.