DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development and Implementation of a Reusable Web-API for Online 3D Reconstruction

Moritz Krüger

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development and Implementation of a Reusable Web-API for Online 3D Reconstruction

Entwicklung und Implementierung einer wiederverwendbaren Web-API für Online 3D Rekonstruktion

Author:	Moritz Krüger
Supervisor:	Prof. Gudrun Klinker, Ph.D.
Advisor:	Linda Rudolph, M.Sc.
Submission Date:	15.08.2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2021 Moritz Krüger

Acknowledgments

Firstly, I would like to thank my advisor, Linda Rudolph, for her excellent support and skilled expertise throughout this thesis. Moreover, I would like to especially thank her for the help during the writing stage of this thesis.

I would like to extend my sincere thanks to Katharina Aichinger and Stephan Krüger for their omnipresent helpful advice, practical suggestions, and continuous encouragement.

Lastly, I would like to express my deepest gratitude to my parents for their continuous emotional and financial support throughout my Bachelor's studies at TUM.

Abstract

In this thesis we propose an architecture for a reusable web service for online 3D reconstruction. The purpose of this service is to enable developers and end-users to perform reconstructions, compare photogrammetry software, and support the future development of photogrammetric applications by offering all functionalities through a single reusable web-API. The main focus point of this work is the reusability aspect of the web service architecture. We introduce a decentralized and containerized microservice structure that allows new components and photogrammetry software to be integrated into the system easily. Our web service provides a platform to manage image-sets and image metadata as well as a standardized way of starting and parameterizing reconstruction pipelines. Further, we implement a way of using existing camera poses during the reconstruction process. The system is evaluated from structural and functional viewpoints and it is shown that our proposed web service presents a reusable and well maintainable approach for web-based reconstruction and lays the structural foundations to support further development of our service in the future.

Contents

1. Introduction

1.1. Motivation

3D reconstruction is a broad term that finds its main application in the computer vision and computer graphics field. The derivation of a 3D model from the shape and appearance of an real-life object or scene represents the main task of 3D reconstruction. A vital research domain in this field is *photogrammetry*. It includes image-based 3D reconstruction approaches which use images from multiple viewing angles to obtain a 3D model.

Deducing spatial information to reconstruct the structure of visual features from 2-dimensional photographs has been a demanding challenge for a prolonged period of time and is essential to numerous applications. Besides a wide range of possible applications, the most commonly used fields of application include object identification, robot navigation, scene understanding, industrial control, 3D modeling and animation, and medical diagnosing. Algorithms to recover the third dimension from two-dimensional images have been thoroughly researched for several decades. [HLB19, p. 1]

In recent years the number of areas 3D models are being used in has been increasing rapidly, leading to a strong rise in the demand for 3D content. Whether used to conserve cultural heritage, depict real-life scenes in video games or movies, or to create models for 3D printing or AR applications, models generated through the use of 3D reconstruction software are omnipresent in today's world. With it, requirements for photogrammetry software have become more variant, depending on its field of application.

As of now, there are several photogrammetry software solutions available. Most notably: Agisoft Metashape, AliceVision Meshroom, COLMAP, OpenDroneMap and 3DF Zephyr. Despite the vast pool of available software, there are few fully automated reconstruction pipelines available and many steps still need to be performed manually by the user. Comparing different photogrammetry tools is a time-consuming task as local installations are required, images need be to imported into numerous applications multiple times, and interactions with various photogrammetry graphical user interfaces (GUIs) are necessary. Sharing, testing, and comparing newly developed 3D reconstruction approaches represents another laborious task since code has to be shared and built on another system. Further, there exists no standardized way of starting reconstruction tasks. In addition, improved wireless connectivity and the rise of cloud-based services have sparked interest in the research and development of cloud- or web-based online 3D reconstruction services. For those reasons, the development of a reusable web-API for online 3D reconstruction is required.

1.2. Contributions

In this thesis, we propose a concept for an extendable web-based server for online 3D reconstruction providing a reusable web-API. Our main contributions include:

- A reusable and RESTful API suitable for different client applications
- A web-based service capable of handling the upload and management of image-sets and performing reconstructions easily
- An extendable and maintainable server that allows for simple integration of different photogrammetry software
- An architecture that helps develop, share, test, compare and utilize 3D reconstruction software

1.3. Thesis Structure

The thesis is structured as follows. To begin, we examine the fundamentals required to understand concepts and implementations that were put to use in our application in Chapter 2. After that, we discuss related work in this field in Chapter 3 in regards to similar existing web services and existing cloud reconstruction services related to our proposed system. In Chapter 4 we describe our proposed web server architecture and its implementation. After having discussed the development and implementation aspects, we evaluate our system in Chapter 5 using a brief structural and functional evaluation assessing if the operative and architectural properties of our web service are as expected. In Chapter 6 we critically discuss certain aspects of our web service architecture and implementation. Finally, Chapter 7 concludes this thesis with a short summary and an outlook on future work.

2. Fundamentals

In this chapter, we discuss fundamentals that form the basis of our project. We start by examining the basics of photogrammetry and introducing different photogrammetry software that were integrated into our system before introducing the microservice architectural pattern which was also implemented in our proposed architecture. Lastly, we explore the foundations of containerization technology and define terms and definitions required as prerequisite knowledge.

2.1. Photogrammetry

In order to get a rough understanding of the technologies used in this thesis, photogrammetry, the photogrammetry workflow, as well as the used photogrammetry software have to be introduced.

Though there is no universally accepted definition of photogrammetry, it can in essence be described as "the science of obtaining reliable information about the properties of surfaces and objects without physical contact with the objects, and of measuring and interpreting this information" [Sch05, p. 3]. Other definitions may be more specific, characterizing it as the "science of measuring in photos" [Lin09, p. 1], directly associating it with the use of image information.

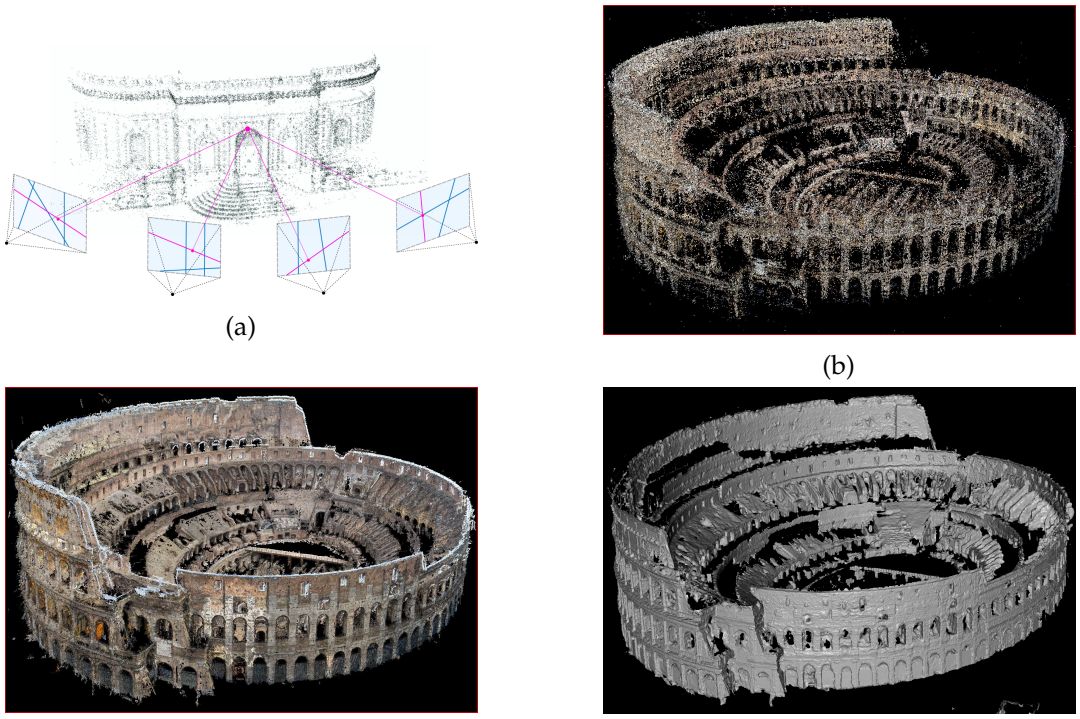
2.1.1. Basic Photogrammetry Workflow

For most applications, the basic photogrammetry workflow is divided into seven steps, though deviations are possible. These steps will be discussed briefly in the following:

Image Acquisition Every reconstruction task starts with the acquisition of images. This is typically done by collecting multiple photos of the object or scene from various angles and grouping them into an image-set which will be the basis for the reconstruction process. Furthermore, extra reference information and scale metrics should be collected and correct exposure of the images needs to be ensured. For outdoor environments, the weather is an important factor. Wind, rain, or other weather conditions causing objects to move or cause vast changes in the lighting conditions can lead to poor reconstruction results and should therefore be avoided. [Lac17] Lastly, it is recommended to have a good amount of overlap in the images.

Feature Extraction In order to find prominent elements in the images feature keypoints need to be extracted from the images. These can later be matched to one another, as a means of finding 2D correspondences. Popular feature descriptors are SIFT (scale-invariant feature transform) and SURF (speeded up robust features).

Feature and Image Matching In this step common points in images and images looking at similar areas of the scenes are identified. This is commonly done by matching feature descriptors using different matching methods. These 2D matches provide the basis for the reconstruction process.



(a) (b) (c) (d)

Figure 2.1.: (a) shows an illustration of the SfM process, finding a common point in four images and re-projecting them into the scene to estimate the 3D location. (b) shows an example sparse point cloud, whereas (c) shows the dense point cloud after dense reconstruction. Finally (d) shows the corresponding untextured mesh. Images taken from [Gep+20] and [Aga+10].

Sparse Reconstruction After getting information about 2D correspondences in the image, the 3D locations of these points are estimated. This can be done by using SfM (structure from motion) algorithms, yielding a sparse point cloud of the object or scene. SfM algorithms also approximate the camera parameters, i.e. camera intrinsics and extrinsics, for every image, besides the set of 3D points [FHP15]. Figure 2.1a illustrates the SfM process and Figure 2.1b

shows an exemplary sparse point cloud.

Dense Reconstruction In the dense reconstruction step, the sparse point cloud is used to compute depth information. This is done in the form of estimating depth and normal maps for each image. This is then used to create a dense point cloud. Though this dense cloud may already resemble a "solid" surface from certain angles, it is still just an aggregation of single pixels [Sac+20]. A dense point cloud can be seen in Figure 2.1c.

Meshing At this stage, a 3D mesh can be created using the depth and normal information of the dense point cloud. An example can be found in Figure 2.1d.

Texturing The produced mesh can be textured using different texture generation approaches. In general, these methods use the original images and information from the previous steps, i.e. camera intrinsics and extrinsics, as for instance proposed by Baumberg.

2.1.2. Used Photogrammetry Software

For our application, we integrated several photogrammetry software. This section aims to give a short overview of the used technologies.

AliceVision Meshroom

Meshroom[1] is a free and open-source 3D Reconstruction Software based on the AliceVision framework [Alia]. AliceVision[2] is a photogrammetric computer vision framework, which provides 3D Reconstruction and Camera Tracking algorithms [Alib]. Meshroom supports the execution of a basic photogrammetry pipeline via the command-line interface which is essential for the integration into an automated pipeline.

COLMAP

COLMAP[3] is an open-source, multipurpose SfM and Multi-View Stereo (MVS) pipeline. It includes a graphical and command-line interface and offers a broad range of functionalities for the reconstruction of organized and unorganized image-sets. [Scha] COLMAP is based on [SF16] and [Sch+16]. COLMAP offers multiple command-line interface commands and can therefore be integrated into an automated pipeline easily. However, it does not support texturing. Thus, OpenMVS was partially substituted in COLMAP's dense reconstruction steps in order to provide textured models.

[1] https://alicevision.org/#meshroom
[2] https://alicevision.org/
[3] https://colmap.github.io/

OpenMVS

OpenMVS[4] is an open-source library for MVS reconstruction. It addresses the last part of the photogrammetry chain-flow and provides a full set of algorithms to reconstruct the complete surface structure of objects. It includes steps for dense point-cloud reconstruction, mesh reconstruction, mesh refinement, and texturing. [Cer20]

It was integrated, due to partially missing texturing features in COLMAP.

2.2. Microservice Architecture

In this thesis, we develop a web service composed of multiple internal web microservices based on the microservice architectural pattern. For that purpose, the following section will focus on the basics of the microservice architecture and what differentiates it from a monolithic architecture.

2.2.1. General Concept

In a microservice architecture, tasks get split up into smaller, modular components, each being handled by a microservice. Communication with the client or other microservices takes place through lightweight REST API requests. Microservices enable the design of applications as a collection of loosely coupled services, which provides easy scaling capabilities and further supports the maintainability of the system. [Ora21]

2.2.2. Benefits of a Microservice Architecture

In order to understand the advantages of using a microservice architecture, one has to understand how this approach differs from a traditional monolithic architecture.

In a monolithic architecture, the whole system consists of a single component capable of performing each task. Figure 2.2 illustrates the monolithic and microservice architecture on the example of an online shop. There are many key differences in the characteristics of the two architectures, a list of which can be found in Table 2.1.

The further motivation behind the utilization of the microservice architectural pattern for our web-based application will be discussed in further chapters of this thesis.

2.3. Containerization

In the implementation of our proposed system, we utilize containerization technology in combination with the aforementioned microservice architecture. Hence, in this section, we examine the general concept of containerization, its benefits, and synergies with the microservice pattern.

[4] https://github.com/cdcseacave/openMVS

Characteristic	Microservice Architecture	Monolithic Architecture
Unit design	The application consists of loosely coupled services. Each service supports a single business task.	The entire application is designed, developed, and deployed as a single unit.
Functionality reuse	Microservices define APIs that expose their functionality to any client. The clients could even be other applications.	The opportunity for reusing functionality across applications is limited.
Communication within the application	To communicate with each other, the microservices of an application use the request-response communication model. The typical implementation uses REST API calls based on the HTTP protocol.	Internal procedures (function calls) facilitate communication between the components of the application. There is no need to limit the number of internal procedure calls.
Technological flexibility	Each microservice can be developed using a programming language and framework that best suits the problem that the microservice is designed to solve.	Usually, the entire application is written in a single programming language.
Data management	Decentralized: Each microservice may use its own database.	Centralized: The entire application uses one or more databases.
Deployment	Each microservice is deployed independently, without affecting the other microservices in the application.	Any change, however small, requires redeploying and restarting the entire application.
Maintainability	Microservices are simple, focused, and independent. So the application is easier to maintain.	As the application scope increases, maintaining the code becomes more complex.
Resiliency	The application functionality is distributed across multiple services. If a microservice fails, the functionality offered by the other microservices continues to be available.	A failure in any component could affect the availability of the entire application.
Scalability	Each microservice can be scaled independently of the other services.	The entire application must be scaled, even when the business requirement is for scaling only certain parts of the application.

Table 2.1.: Table summarizing the central differences between the microservice architecture and monolithic architecture. Table directly taken from [Ora21].

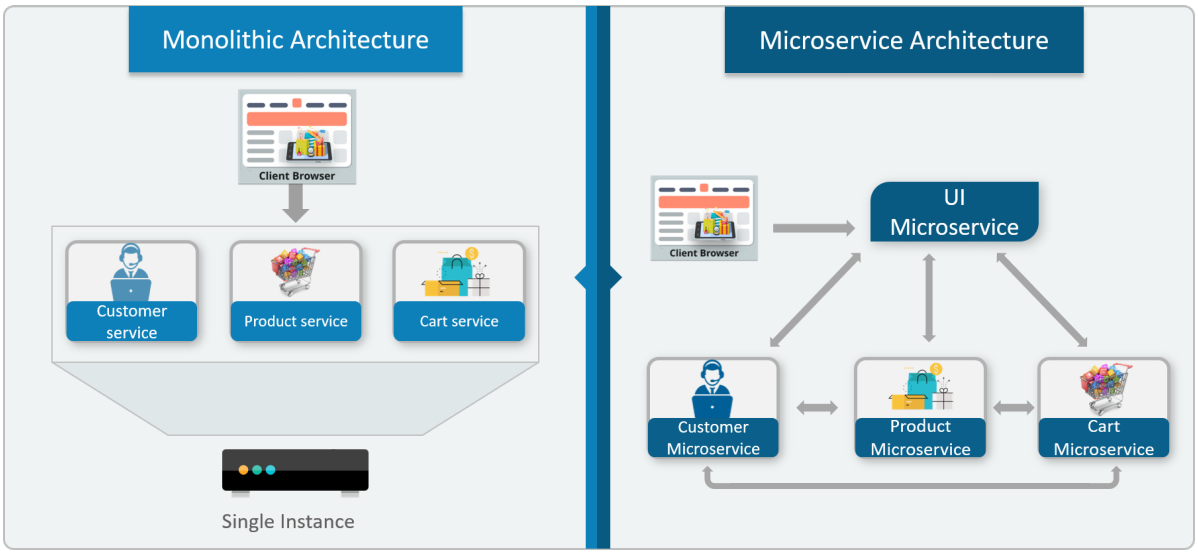


Figure 2.2.: Illustration of the monolithic (left) and microservice architecture (right) on the example of an online shop. The left side shows the main application as a single instance app, internally divided into multiple services. On the right side, the main application is divided into multiple microservices. Further, intercommunication between the microservices is represented by gray arrows between the services. Image taken from [Kap20].

2.3.1. General Concept

Containerization defines the concept of packaging applications into small lightweight executable containers. It bundles the software code, operating system (OS) libraries, as well as other code dependencies into a single unit, that can be executed continuously on any infrastructure. Though the general concept of containerization and isolation is not completely new, it first started seeing usage in 2013 utilizing the Docker Engine[5], an open-source and now industry standard for universally packaging applications. A main characteristic of a container is its lightweight nature. This can be attributed to the fact that containers do not necessitate overhead in the form of linking the application's OS with the host OS as they also utilize the very same OS kernel as the host system. This is achieved by use of the container runtime responsible for the actual execution of the containers. A key benefit of this structure is the increased portability as applications can be "'written once and run anywhere'"[Edu21]. [Edu21]

2.3.2. Distinction to Virtual Machines

Virtual machines (VMs) enable the execution of several operating systems on a single machine. For that purpose, the entire operating system, as well as the application dependencies, are

[5]https://www.docker.com/

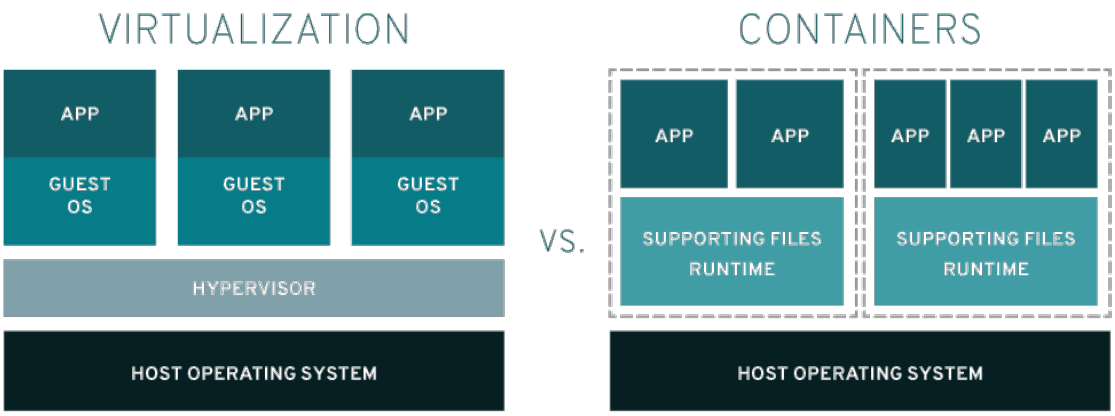


Figure 2.3.: The left side shows a containerized structure, whereas the right side shows the structure of virtual machine-based applications. Note that no full copy of the utilized OS is required in the containerized structure, in contrast to the VM architecture. Image taken from [Hat].

packaged into a virtual machine. [Edu21]

As opposed to the container runtime, VMs use a hypervisor that segments the physical resources that will later be partitioned and allocated to each virtual machine [Hat].

Even though containers are in a way comparable to virtual machines as they have similar security benefits due to the isolation of applications, the former has some crucial benefits over the latter. The above-mentioned overhead is significantly less as containers do not require a full copy of the OS and can therefore use resources more effectively. Another advantage is that containers are innately smaller in size. The enhanced portable packaging allows containerized applications to be deployed and used across any cloud or platform. To sum up, it can be stated that containers are more portable, efficient, and flexible than virtual machines. [Edu21] Figure 2.3 illustrates an exemplary structure of containerized and virtual machine-based applications.

2.3.3. Benefits of Containerization

In the following, we summarize the beneficial characteristics of containerization from a more generalized perspective.

Portability As previously mentioned the containerized structure allows for greater portability, due to its compact packaging and ability to be deployed in a wide range of environments.

Speed and Efficiency The remarked lightweight nature, little overhead, and compact size of containers allow for greater execution speeds and increased efficiency.

Fault Isolation As containers are isolated from the host system and other containers, a container's failure does not influence the execution of other containers. [Edu21] Hence, other parts of the system's functionality are unaffected.

Management Simplicity Containers can be orchestrated with the help of container orchestration platforms. These platforms can provide more benefits including scaling and monitoring of containerized applications. The most notable container orchestration system is Kubernetes[6]. [Edu21]

Security Potentially by malware infected containers can not affect other containers or the host system, as both systems are inherently isolated from each other. [Edu21]

2.3.4. Synergies With Microservice Architectures

Monolithic, as well as microservice structures, are both suited to be used with containerization technology. However, great synergy effects are inherent in a combination of a microservice architecture and containerization technology in such a way that microservices gain all of the containerization benefits. Both the microservice architecture and containerization technology try to divide an application into smaller, more modular components, and thus the goals of both architectures are similar. [Edu21]

Commonly, a single microservice is embedded into a single container, though at times multiple apps can also be run in a single container, e.g. in the form of an app and its related database.

As of today, utilizing microservices encapsulated in containers is common practice in web development and other distributed systems. Thus, in this thesis, we will also implement a combination of both technologies. Further details will be presented in later chapters.

2.4. Terms & Definitions

This section introduces frequently used terms and definitions as well as prerequisite knowledge in relation to web-based systems that aid the understanding of our proposed service.

2.4.1. Client-Server Architecture

The Client-Server architecture is widely used as a form of communication in a network-based environment. A client can be described as "a system or a program that requests the activity of one or more other systems or programs, called servers, to accomplish specific tasks" [Han00, p. 3]. In our context, a client can be a mobile device, another computer, or even another server requesting our resources through the use of the API. A server can be characterized as "a system or program that receives requests from one or more client systems or programs to perform activities that allow the client to accomplish certain tasks" [Han00, p. 3]. In the

[6]https://kubernetes.io/

client-server workflow the server processes said received request and sends a response back to the client. This interaction is made possible by the use of the API.

2.4.2. API

As described by Reddy, "[a]n *Application Programming Interface* (API) provides an abstraction for a problem and specifies how clients should interact with software components that implement a solution to that problem" [Red11, p. 1]. Usually, in web-based development, the software component that implements the solution is the server and is accessible through the web-API. Web-APIs act as the main layer of interaction between client and server and are used to provide the server's functionalities and services to client applications. This is commonly done by utilizing HTTP requests. Further, there exists the concept of RESTful APIs. For this type of API, resources can be requested by making use of standardized HTTP methods, namely GET, POST, PUT, DELETE, and other RESTful methods [SR19]. Requests are stateless, meaning that a single request contains all required information to perform the requested task [Cas07].

2.4.3. Web Server, Web-API, and Web Service

A web server is a particular instance of a server and provides its functionalities through web-based methods. The terms web server and web-API can often be confused, as their functionalities are closely interwoven and the functionalities of the web-API also cascade to the web server. However, we want to remark that per definition the web-API is the interface that allows client and server, in this case the web server, to communicate in a standardized way. To group both terms we can use the term *web service*. A web service is a web server that provides its functionalities through the web-API.

2.4.4. Web Service Reusability

Our goal is to design a reusable web-API for 3D reconstruction. This reusability facet also cascades to the implementing web server as we also want to further develop and reuse our web server. Hence, we ought to describe the concept of reusability in the context of web-based services.

General Concept Fundamentally, the term *reusability* describes the idea of reusing a web service for multiple purposes. This entails that as opposed to creating a new client-specific API for every new client the same API is used each time. This increases maintainability as only one generalized API has to be actively maintained. Benefits include fast development and simple integration of new clients. To allow new functionalities to be implemented easily and existing code to be reused, the implementing web server should be extensible and maintainable. As it is intended for our web service to be utilized by multiple client applications and extended further in the future, reusability is an important aspect of our development process.

Required Characteristics We define the following criteria a web service needs to meet in order to comply with our definition of reusability:

- **Generalized Functionalities** The web-API needs to provide generalized functionalities that are not limited to the use case of a single client. As remarked, this allows for multiple reuse of the API by various client applications.
- **Documentation** The web service needs to provide extensive documentation for the usage of the API routes to client applications and future developers to help client-side integration and later development.
- **Extensibility** In order to be reused continuously, a reusable web service has to be extensible. The addition of new features and functionalities has to be straightforward and uncomplicated.
- **Maintainability** The reusable web service needs to be maintainable meaning it should be easy to implement code changes and feature adjustments.

3. Related Work

In this chapter, we discuss similar publications related to the development of a web-based API for 3D reconstruction and explore existing cloud-based solutions.

3.1. Related Publications

In 2015, Heller et al. developed an SfM web service aimed to provide access to SfM methods developed at the Center for Machine Perception of the Czech Technical University in Prague. The authors propose a web service allowing users to access and execute different photogrammetry "jobs". They offer different job types ranging from camera calibration, different sparse and dense reconstruction algorithms to so-called "One-button" methods, in essence full photogrammetry pipelines. They also integrated third-party photogrammetry software in the form of Bundler[1]. [Hel+15]

Unfortunately, the mentioned publication does not go into detail about the exact structure and implementation of the system and merely presents the features and capabilities of the service. Also, there is no source code publicly available. The goals of this thesis are partially overlapping with the mentioned publication as they too develop a web service for online 3D reconstruction, aimed at the execution of reconstructions and also developmental aspects. The *task-description* that will be introduced in this thesis is inspired by the custom job XML files proposed by Heller et al.

A key difference to this publication is that our thesis is further aimed at the development of an extensible and maintainable web server, giving insight into the structure and implementation of the proposed system. Further, we are more focused on comparing different photogrammetry software and the integration of our service by a wide variety of client applications using our reusable web-API. Lastly, our system is not meant to be limited to SfM algorithms, but to house other 3D reconstruction methods in the future as well.

3Dnow is another web-based 3D reconstruction service. It was proposed by Tefera et al. in 2018 and utilizes the open-source photogrammetry software COLMAP. Its main innovative aspects include providing a fully automatic web-based photogrammetry pipeline as well as a fully automated cleaning process of the output meshes and point clouds. It includes parameter tuning capabilities, scaling and georeferencing of point clouds, an objective evaluation schema of sparse point clouds, and detailed reconstruction metrics. 3Dnow is accessible through a standard web browser. It uses a decentralized service structure that is accessed through a central request manager. It is implemented using a Python-based web framework and utilizes queue-based task scheduling. [Tef+18]

[1] https://www.cs.cornell.edu/ snavely/bundler/

There is no explicit mention of a dedicated web-API for integration by different client applications, so it is assumed to be used using its provided browser interface which is the first distinction to be made in regards to our thesis. Another main difference between 3Dnow and our proposed system is that our web service is designed to be extended and used to compare different photogrammetry software. Also, there is no source code available for review, hence we can not fully understand its implementation and architectural decisions made by the authors. 3Dnow offers a lot of functionalities that could also be beneficial in our system and the proposed architectural structure seems to be a valid approach. In our thesis, we will also implement COLMAP, a comparable decentralized service architecture, and task-scheduling technology.

3.2. Existing Cloud-Based Reconstruction Services

As of now, there are a few existing cloud-based reconstruction services available. This section will introduce some of the most popular services and outline the main differences to this thesis.

Autodesk ReCap Photo Autodesk ReCap Photo[2] is a cloud-based service and is based on Autodesk ReCap Pro. It provides reality capture, 3D scanning, and intelligent model creation [Aut]. It offers a lot of advanced features such as georeferencing and mesh editing tools.

Agisoft Cloud Agisoft Cloud[3] is another cloud platform specialized in photogrammetric reconstruction tasks. It is based upon Agisoft's standalone photogrammetry software Agisoft Metashape and offers photogrammetric reconstructions, areal LIDAR workflows, and many more features [LLC20].

WebODM Lightnight & PIX4Dcloud Services primarily focused on drone image processing include WebODM Lightning[4] and its main competitor PIX4D[5]. WebODM Lightning uses the open-source software OpenDroneMap, whereas Pix4D uses its own proprietary software solutions. Both provide tools for cloud-based photogrammetric reconstruction for aerial images.

There are many key distinctions between the aforementioned, already existing cloud solutions and the web service proposed in our thesis. Firstly, the mentioned cloud solutions are consumer-focused, meaning their goal is to provide software solutions to the end customer. Yet, our thesis does not only aim to provide a tool for the execution of 3D reconstruction tasks but is also directed at the research and development aspect of photogrammetric software. Our goal is to develop an extensible web service capable of integrating multiple other software

[2]https://www.autodesk.de/products/recap/overview
[3]https://www.agisoft.com/
[4]https://webodm.net/
[5]https://www.pix4d.com/

solutions. Another major difference is that the exact implementation and structure of these cloud-based systems are unknown and, with the exception of WebODM Lightning, use their own proprietary software. Lastly, all of the mentioned systems are paid services, while we intend the development of a free and open-source based web service. The main similarity lies in the features and functionalities we wish to provide. The mentioned platforms offer a lot of advanced and powerful tools for 3D reconstruction that would also be advantageous for our service. Hence, in conclusion, it can be emphasized, that the main purpose of these applications differs vastly from the goals of this thesis.

4. Web Service

In this chapter, we outline the development challenges of the system and introduce the server architecture of our proposed system, its provided functionalities, followed by the made design decisions and the implementation of the web service.

4.1. Development Challenges

In this thesis, we want to propose a system that is capable of managing image-sets, and their associated metadata, perform reconstruction tasks with different photogrammetry software and comply with our definition of web service reusability. Hence, in the first step, we identify the main challenges in the development of our web service.

Image-Set and Image Management It is required to model the relationship of image-sets, images, and its associated metadata in a way that allows the user to manage image-sets and its metadata easily and intuitively.

Integration of Multiple Photogrammetry Software The next challenge is the integration of multiple photogrammetry software. We need to develop a structure that can perform the needed tasks but also satisfies the reusability aspect of our web service. The reusability aspect is the most demanding prospect of the development process as the entire service has to be maintainable and extensible. This presumes that a standardized way of extending the service and utilizing photogrammetry software is found.

Task Management and Execution Though the used photogrammetry software can be used very differently, it is required to find a common method of managing and executing reconstruction tasks. This also means that our image-sets and metadata have to be in a structure that allows them to be used or at least modified effortlessly in a way that makes them usable for multiple applications.

Reusability and Future Development As it is intended for this application to be developed further and continuously reused in the future, special requirements for the structure of the entire system are set. Therefore, we must lay the structural foundations for future features and provide technological flexibility to achieve them in the future.

4.2. General Concepts

In response to the remarked development challenges, we introduce the following general concepts.

Project To model the relationship between images and image-sets, we introduce the concept of *projects*. A *project* in our web service is the entity we use to group images into image-sets. It consists of a project name and project owner. The project name could be the object or scene that the related images show. The project owner was added to associate the images with an owner or the person who created the project. The images can then get uploaded to a given project.

Image The uploaded *image* model consists of the image file as well as its metadata, including camera pose, camera intrinsic information, depth image, and the CPU image (an unprocessed version of the image). These images get linked using the *image identifier*, currently in the form of a common filename. The CPU image was added in response to requirements made by a potential client application.

Provider A *provider* in our web service refers to a single or a collection of reconstruction software used in our web service to perform a reconstruction pipeline or task. We define common API endpoints a provider has to offer in order to be integrated into our system, such that new providers can also be integrated using the very same endpoints. We integrated the photogrammetry software COLMAP, in combination with OpenMVS, and AliceVision Meshroom into our system. Hence, we can define the providers Meshroom and COLMAP.

Task The providers are used to execute tasks. A *task* in our web service represents a 3D reconstruction task. It is associated with a project and provider and executes the reconstruction on the images of the given project.

Task-Description The task execution can further be refined by the optional *task-description*. Each provider can specify different functionalities through the use of task-descriptions. A task-description consists of the task type and task parameters. The providers are free to define the task types and parameterization they offer. These tasks in combination with the task description will be used by the provider in order to execute the selected task with the given parameters.

4.3. Server Architecture

This section aims to give an overview of the server architecture, describe its components and how they interact with each other.

4.3.1. Server Components

The web service is divided into three web-based microservices and seven components in total. An overview of the service structure can be seen in Figure 4.1.

Backend-Manager-Service The first microservice is the `Backend-Manager-Service`. Its main purpose is the management of project and task data as well as the dispatching of tasks to the providers. Further, it acts as the main entry point for client applications.

Central Database The central database is used to store metadata for projects and tasks. The project data includes the project id, the project name and owner, as well as the creation timestamp. The task data contains the task id, the associated project id and provider, the task status, the creation timestamp, the start- and end-time, as well as further metadata that can be filled by the providers.

Message Broker We use a task queue to schedule the reconstruction tasks. This is done using the message broker. The message broker manages a message queue from which applications can enqueue and dequeue tasks.

Meshroom- and COLMAP-Service The remaining two web microservices are the `Meshroom-Service` and `COLMAP-Service`. These services manage the task metadata and schedule the task execution. Their reconstruction capabilities are accessed through the `Backend-Manager-Service` that dispatches the client's requests to the respective provider.

Meshroom- and COLMAP-Worker The execution of the requested task is done by the `Meshroom-Worker` and `COLMAP-Worker`. These worker components dequeue tasks from their respective task queue and execute the given task.

4.3.2. API Structure

Figure 4.2 gives an overview of the existing API routes and how they are related. A full specification of the API can be found in Appendix A.

4.4. Provided Functionalities

With the given server and API structure, our service provides a wide variety of functionalities. We differentiate between the basic functionalities, that were among the initially required capabilities of our service, and the additional functionalities, that were added during the further development of the web service.

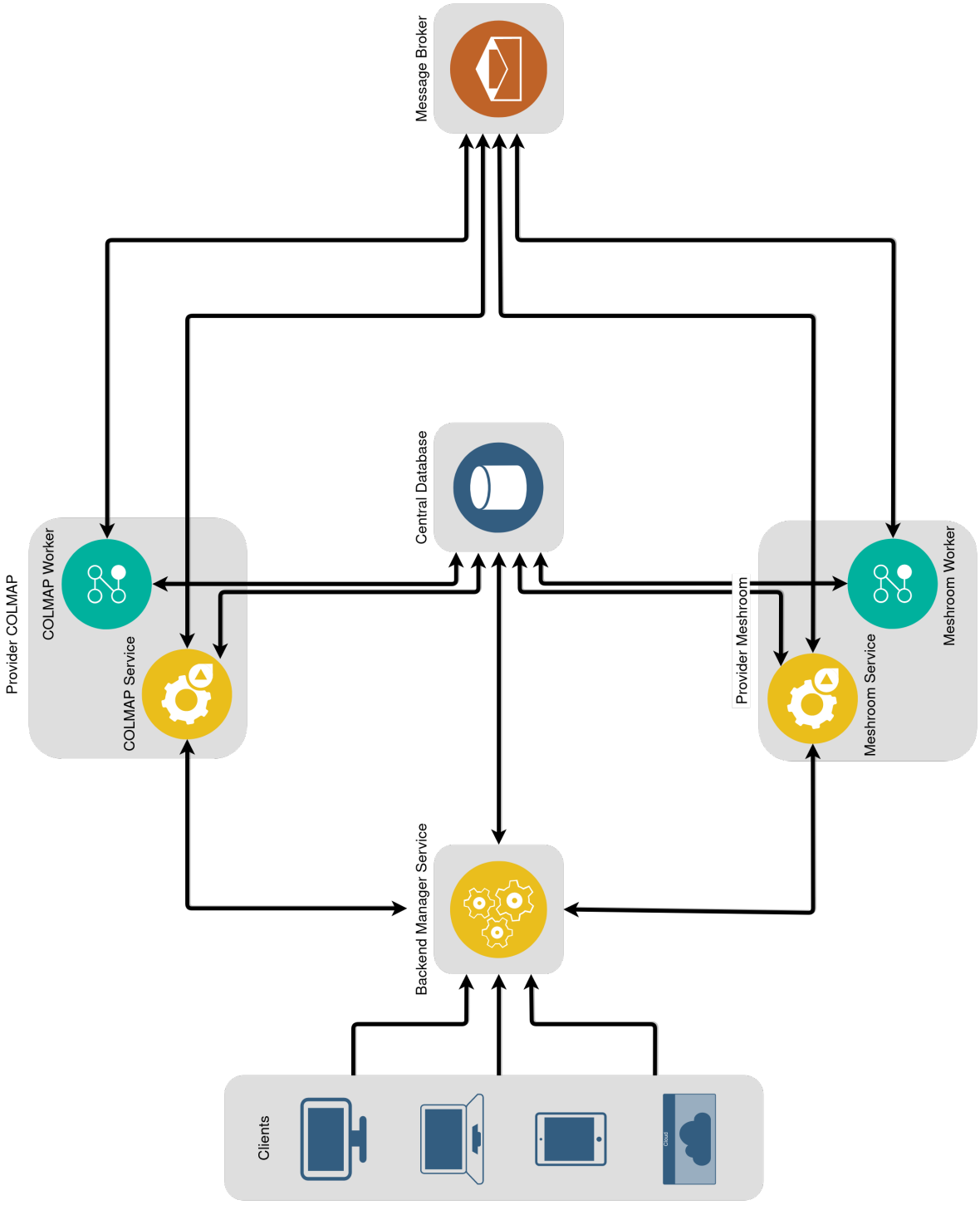


Figure 4.1.: Illustration of our proposed server architecture.

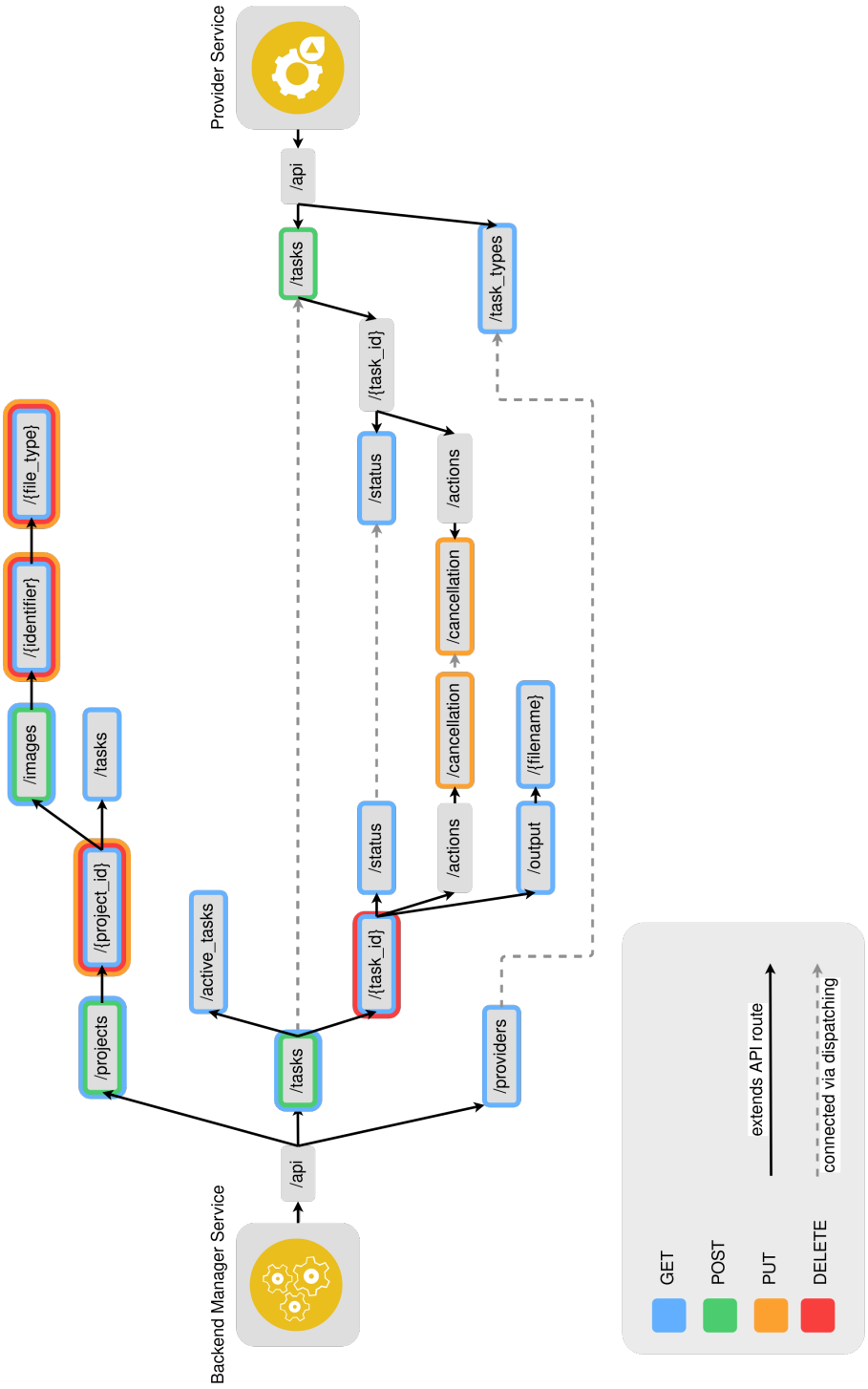


Figure 4.2.: Illustration of the API structure. Note that the `file_type` refers to the metadata of the image and can be the 'image', 'depth_image', 'camera_pose', 'camera_intrinsics', or 'cpu_image'.

4.4.1. Basic Functionalities

Our service is capable of performing the following basic functionalities:

BF1 **Image-sets:** Image-sets can be created.

BF2 **Images and metadata:** Images can be uploaded to the specified image-set. In addition, image-related information can be added. This includes the camera pose as well as the related depth image.

BF3 **Image and image-set management:** Image-sets, images, and related metadata can be uploaded.

BF4 **Reconstructions**: Reconstruction tasks can be started with different photogrammetry software on the provided image-set.

BF5 **Output files:** The output files of the finished reconstruction task can be retrieved.

4.4.2. Additional Functionalities

Furthermore, we support these additional functionalities:

AF1 **Image metadata:** The images metadata also includes camera intrinsic information. This was necessary, as our initial plan to refine the model using the camera pose with COLMAP also required the corresponding camera intrinsic to each image.

AF2 **Image and image-set management:** Image-sets, images, and related metadata can further be updated and deleted through corresponding endpoints. This was added to provide more generalized functionalities and management options.

AF3 **Reconstructions**: For the reconstruction tasks, the task type and parameters can be specified. This allows for a more customizable use of the provider pipelines. In addition, after a reconstruction is started, detailed status information about the task and pipeline steps can be queried. Running tasks can be canceled and deleted.

AF4 **Use of additional information:** Our system partially supports the use of camera pose information in the reconstruction for the provider COLMAP.

AF5 **Browser integration**: The web-API can be accessed through a browser interface and provides detailed documentation of the API routes.

4.5. Structural Design Decisions

We made the following design decisions for our web architecture:

4.5.1. Using a Microservice Architecture

We utilize the microservice pattern. We can divide our web-microservices into two main categories: the components responsible for handling the image and metadata management and entities performing the reconstruction tasks, i.e. the providers. The metadata management will be handled by a single microservice, whereas each provider will function as its own microservice. This has the following benefits, also keeping in mind the properties from Table 2.1:

Increased Maintainability As microservices are independent of each other the applications can also be developed autonomously. This is especially helpful for the development of the providers. In essence, no knowledge of the other systems is required, as we define a given set of endpoints that need to be implemented in the provider microservice. Further details will be discussed in the implementation section.

Technological Flexibility It is possible to develop every microservice utilizing a different technology stack. A technology stack consists of programming languages, frameworks, and other tools and technologies required to develop an application. This enables us to integrate photogrammetry providers that are for instance native C++ applications or offer no command-line interface. These can then be integrated using a C++-based or another alternative implementation of a microservice.

Deployment and Scalability The created microservices can be deployed and scaled separately. Note that though we currently do not utilize scaling of microservices, it is supported in future development by use of the microservice architecture.

In summary, using a microservice pattern is perfect for our use case, as this structure allows for new providers to be integrated easily using its own microservice. Further, it aids the future development of the web service because of its modular structure as well as its maintainability and flexibility aspects.

4.5.2. Using Containerization Technology

As stated before, containerization works great in combination with a microservice architecture. The following properties of containerization were especially important factors for its integration into the system:

Portability As this platform is still in development, later development may take place in different environments. Using containerization allows us to deploy our system in a wide variety of environments easily, whether local or in the cloud. This relieves much of the work associated with setting up the system and its dependencies, aiding further development.

Fault Isolation The failure of a single provider will not influence the stability of the system, which means that during the integration of new providers, potential errors or instabilities will not affect the reliability of the rest of the system.

Use of Container Orchestration Systems In the future, the use of a containerization orchestration system is feasible. This could provide horizontal and vertical scaling capabilities.

4.5.3. Provider Interface

To incorporate the providers uniformly into our system, we have set requirements in the form of API-routes the providers have to implement in order to get integrated into our web service. Thereby, we can further increase the maintainability and extensibility of our system. An illustration of the provider interface can be seen in Figure 4.3.

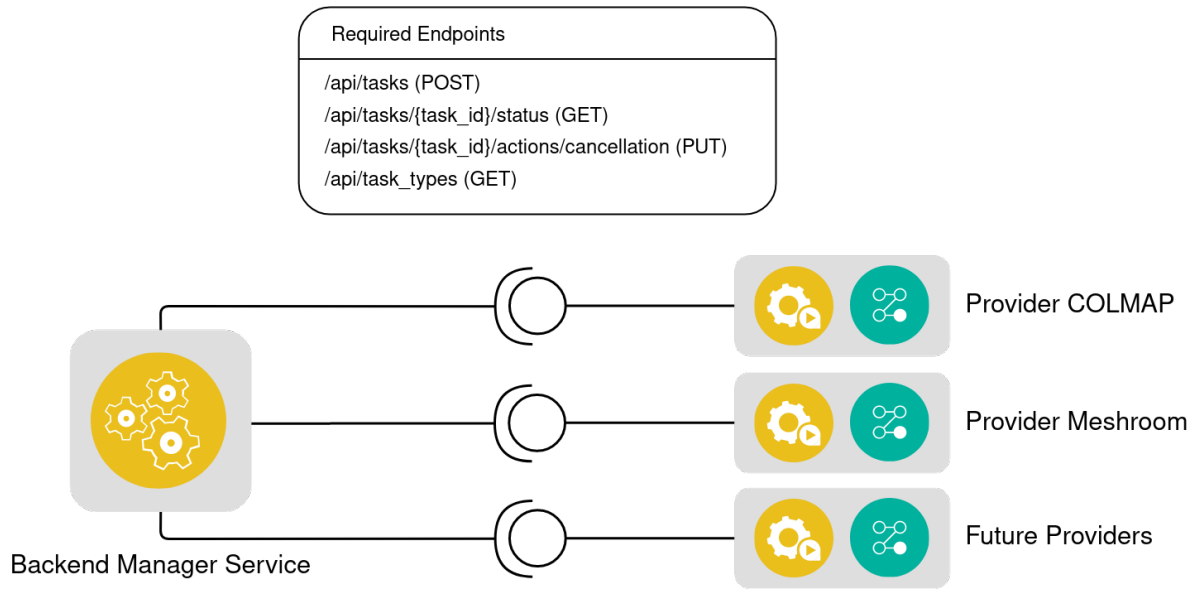


Figure 4.3.: Illustration of the provider interface. The Backend Manager Service uses the required API-endpoints to integrate the providers in a standardized way.

4.5.4. Provider Subdivision

For our providers COLMAP and Meshroom we apply an additional division step. Each provider is divided into a service and worker part. The service part is responsible for receiving the initial task and other provider-specific management aspects, whereas the worker part is in charge of the actual execution of the tasks. The idea behind it is that the worker application can potentially be scaled horizontally and vertically independently from the service application in the future. This division is highly recommended for the integration of new providers but is not compulsory.

4.5.5. Using a Task Queue

The task execution is powered by the use of a task queue. Tasks queues are advisable when resource-intensive background computations or other application blocking tasks are to be performed. In our case, this is given by the long and computationally intense reconstruction tasks. As we do not want our provider services to block during a reconstruction, we instead use the task queue to hand over our tasks to the worker applications. This allows tasks to be scheduled sequentially and executed one by one or in parallel by multiple workers. We utilize the task queue system in both of our providers. Again, the use of the provided task queue system for new providers is strongly advised, but not compulsory. This will give new providers technological freedom to use their own implementations for task management if required. An illustration of the task queue system is shown in Figure 4.4.

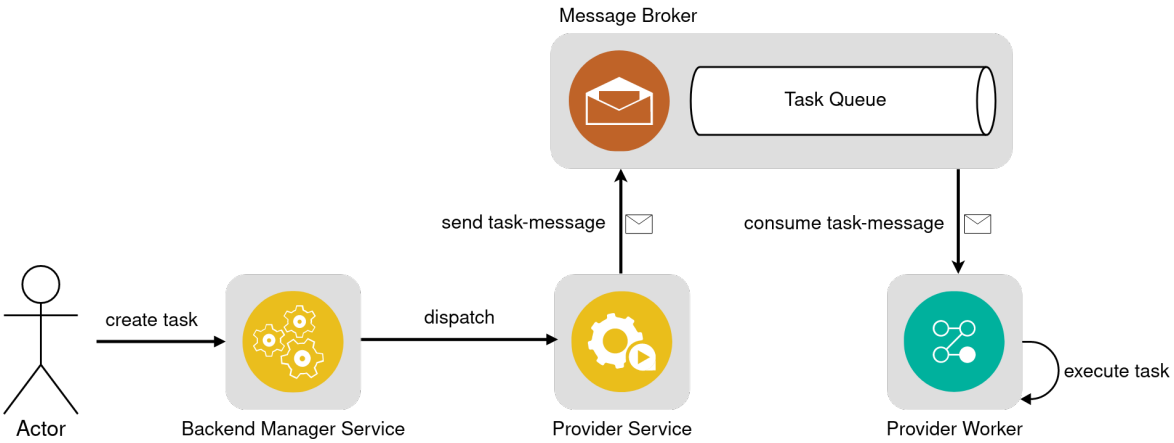


Figure 4.4.: The Backend Manager Service receives the initial request for task creation and dispatches it to the selected provider. The corresponding provider service then schedules the task by enqueuing it into the message queue. The provider worker then consumes said message and starts the task execution.

4.6. Implementation

In this section, we describe the implementation details of our proposed web service. Python is a simple and flexible programming language and offers fast development of web-based applications [TG14]. For that reason, the code was developed, implemented, and tested using Python 3.8.5. The source code is available on GitLab [Krü21].

4.6.1. Design Goals

We have set these design goals for our implementation and codebase:

D1 **Readability:** The code should be understood easily.

D2 **Modularity:** The code should be divided into modular components. Further, the purpose of each component should be intuitive and well defined.

D3 **Modifiability and Extensibility:** Code components should be refactorable and easily replaceable. Moreover, new code segments should be able to get integrated without difficulty.

4.6.2. Code Design

Based on the implementation design goals we also have to discuss structural decisions made for the source code, as it has a big impact on the maintainability and extensibility of the application.

Applied Principles For our source code, we tried to divide the application into modular, intuitive, and well-organized components. Furthermore, we try to use a similar structure in all components in a way that allows future developers to understand each component more easily. For the general code flow, we try to use descriptive function names that make it obvious what the function does exactly and keep functions short whenever possible. We try to use meaningful variable names to improve the readability. Additionally, we try to reduce duplicate code to a minimum. Lastly, it was attempted to integrate dynamic code into the web server whenever feasible, as opposed to hard-coded code.

4.6.3. Used Technologies

Web Microservices For our web microservices components we use FastAPI[1], a framework for building web-based APIs. An alternative option to build web services in Python is Flask[2]. In the following, we will briefly go over why we chose FastAPI over Flask for our application.

The first prototype for our web-based services was utilizing Flask, one of the most popular web frameworks for Python. During early development stages, it was discovered that Flask does not provide an automatized way of documenting API routes. We used the Flask extension Flask-RESTplus[3] that provides more REST support and includes a Swagger UI[4] integration. The Swagger UI is generated from an OpenAPI specification and enables enhanced API documentation and provides a browser interface to view and test API endpoints, with detailed information about the request and response payload. The crucial factor that discouraged the further use of Flask, was the lack of data validation that had to be implemented by hand. Consequently, we decided to use FastAPI.

FastAPI is quite similar to Flask but natively includes a Swagger UI integration and data validation all-in-one. It allows for the creation of complex Pydantic[5] data models that are

[1] https://fastapi.tiangolo.com/
[2] https://flask.palletsprojects.com/en/2.0.x/
[3] https://flask-restplus.readthedocs.io/
[4] https://swagger.io/tools/swagger-ui/
[5] https://pydantic-docs.helpmanual.io/

automatically validated in our API. The interested reader can find a small code example of a pydantic model in Appendix B.

The web microservices are deployed using Gunicorn[6] and Uvicorn[7] which is the recommended way of deployment as described in the FastAPI documentation. The interested reader can find more information about the deployment technologies on the respective website.

Task Queue Our task queue scheduling is implemented using celery[8] and the open-source message broker RabbitMQ[9]. Celery is a distributed task queue focussed on real time-processing. It offers a Python-API to manage the task queue and send tasks to the queue. Further, it processes said tasks in a dedicated worker application. Hence, our provider workers are run using celery and the provider services use the celery Python-API to manage the execution of tasks. Celery further requires a message broker. We chose RabbitMQ, a lightweight message broker, as it is listed as a stable broker for celery [Cel].

An alternative to celery is the Python package 'RQ'[10]. However, due to the various broker support and flexible celery API, we chose celery over 'RQ' for our task queue implementation.

Containerization We package our web application into standardized and shareable units for development and deployment using Docker[11]. The Docker Engine is the most common software used for containerizing applications. Another advantage of using docker is its compose tool[12]. We can define our services, network bridges, and shared volumes using a `docker-compose.yml` file. By doing that, we can build and run our application using only a few command-line calls. Using Docker Compose allows for uncomplicated deployment of our entire application on any machine. Further, it can be used to configure parts of our application such as database passwords, application ports, et cetera.

4.6.4. Upload Process

As before-mentioned, our uploaded image model consists of the image, the depth image, the CPU image, a camera pose file, and a camera intrinsic file.

Image Files The image files are uploaded as-is, meaning that they will not be modified during the process. They are expected to be standard image files, e.g. jpg and png, however, there are currently no validity checks implemented.

[6] https://gunicorn.org/
[7] https://www.uvicorn.org/
[8] https://docs.celeryproject.org/
[9] https://www.rabbitmq.com/
[10] https://python-rq.org/
[11] https://www.docker.com/
[12] https://docs.docker.com/compose/

Camera Intrinsic File The camera intrinsic file is a standard text file in the format: $p_x\ p_y\ f_x\ f_y\ w\ h$, representing the principle point x and y (in pixels), the focal length x and y (in pixels), and the sensor width and height respectively. This information encapsulates the information required to construct a camera intrinsic matrix. The interested reader can find additional information about the camera intrinsic matrix in [Sze10].

Camera Pose File If a camera pose file is uploaded, additional information is required. We further require the *pose format* the *transformation direction* and the *coordinate system*. The pose format specifies if the camera pose is represented as a quaternion + translation string (QT), a 4x4 transformation matrix string (T), or a 3x4 rotation-translation (3D Euclidean transformation) matrix string (RT). In any case, the camera pose file is a standard UTF-8 formatted text file. The transformation direction specifies whether the given pose is a world to camera (W2C) transformation or camera to world (C2W) transformation. Lastly, the coordinate system defines which coordinate system is used. This can be a left-handed (LH) or right-handed (RH) coordinate system. We also offer the the possiblity to upload the camera pose as a 'formatted' text file. A formatted text file contains the required metadata in the first line of the text file, for instance 'QT LH W2C', and the rest of the information in the following lines. Examples for each type can be seen in Listings 4.1-4.4. For the interested reader, we refer to [Sze10] for further information about camera pose representations and transformations.

Camera Pose Handling In order to handle camera pose information we implemented the `camera_poses` package. It provides the `CameraPose` class that can load camera pose data from text files, apply transformations, such as switching coordinate systems or inverting the camera pose, and is used by providers to process the supplied camera pose data into the required format. It is capable of handling data from quaternions, 4x4 transformation matrices, and 3x4 camera projection matrices.

```
-0.433219993583 -0.055553650375 -0.899574471120 3.247106620677
0.056781376558 0.994433571044 -0.088756678939 0.140327149378
0.899497811246 -0.089530244667 -0.427654092525 0.557238856010
```

Listing 4.1: Example camera pose file in 'unformatted' RT format. Each row represents a row in the rotation-translation matrix.

```
-0.433219993583 -0.055553650375 -0.899574471120 3.247106620677
0.056781376558 0.994433571044 -0.088756678939 0.140327149378
0.899497811246 -0.089530244667 -0.427654092525 0.557238856010
0.000000000000 0.000000000000 0.000000000000 1.000000000000
```

Listing 4.2: Example camera pose file in 'unformatted' T format. Each row represents a row in the transformation matrix.

```
1 0 0 0 0.55555 0.66666 0.7777
```

Listing 4.3: Example camera pose file in 'unformatted' QT format (qw qx qy qz tx ty tz).

```
QT LH C2W
1 0 0 0 0.55555 0.66666 0.7777
```

Listing 4.4: Example camera pose file in 'formatted' QT format (qw qx qy qz tx ty tz)

Resulting File Structure The data is accumulated in the project's folder in different subfolders. The entire file structure of our proposed system is shown in Figure 4.5.

4.6.5. Task Execution

Celery Worker The tasks are executed using celery workers[13]. Tasks are Python functions decorated with the celery task decorator. Each provider worker has its own tasks and own task queue. We currently have two tasks: the `meshroom_batch` and the `colmap_pipeline`. They are parameterized with the task id, the project id, and the parameters given by the task-description. For debugging purposes, we added a FastAPI web server for the execution of said task methods, as celery workers can not be debugged easily. This web server can be found in the `debug.py` file in the respective worker folder.

Command Executor As previously remarked, we use the provider's command-line interface in order to perform reconstruction tasks. For that purpose, Python offers the `subprocess` module that can be used to execute command-line commands in a Python-based application. To add an additional abstraction layer to the execution of command-line commands and manage the execution of more complex command pipelines, we implemented the `command_executor` package into our system. The `command_executor` package consists of the classes: `CommandPipeline`, `Executable`, `Command`, and `Function`. The `CommandPipeline` encapsulates an ordered list of Executables that can be added using the provided `append` method and then executed using the `execute` method. The abstract `Executable` can either be a `Command` or a `Function`. Commands enable the execution of command-line commands and utilize the Python `subprocess` module. However, as sometimes more complex steps are required, we added the `Function` class capable of wrapping native Python functions into the `CommandPipeline`. Additional features of this package include the creation of status and log files. The status files are used by the `/api/tasks/{task-id}/status` route whereas the log information is currently only used for debugging purposes.

In further steps the providers can implement more classes inheriting the `Command`, allowing to utilize and parameterize photogrammetry command-line calls easier. The provided classes help integrate new photogrammetry software as command-line pipelines can be implemented in an intuitive manner. Furthermore, duplicate code is reduced as photogrammetry operations

[13] https://docs.celeryproject.org/en/stable/userguide/workers.html

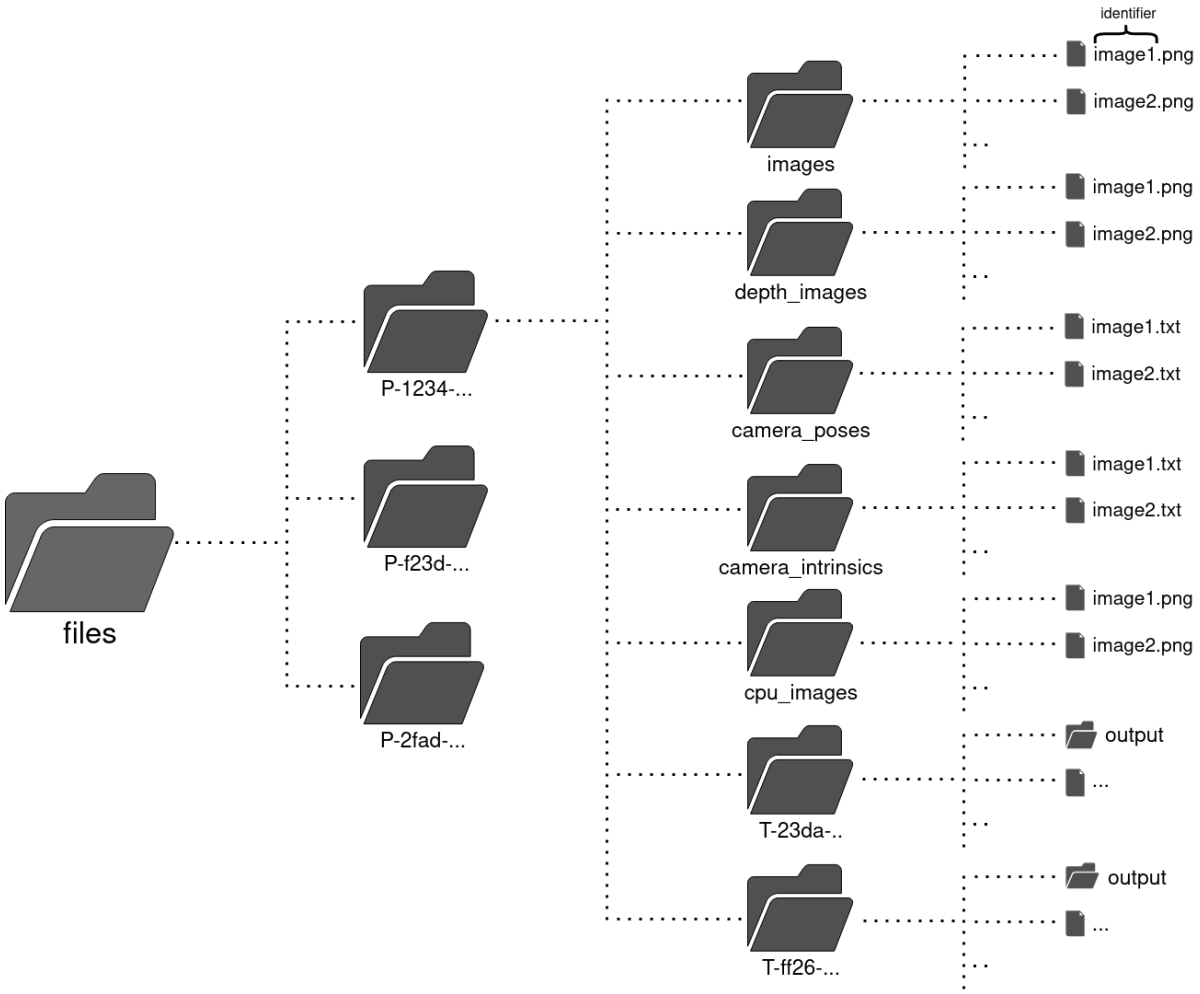


Figure 4.5.: Illustration of the proposed file structure. The files folder is a shared volume mounted into every container that requires file access. The first layer consists of the project folders, referenced by their project id. The second layer contains the image and metadata folders as well as the task folders, also referenced by their id. The last layer are the actual image, image metadata, and task files. Note that for the images and image metadata, related files share the same filename, given by the identifier.

only need to be defined once and can be used by multiple methods. The class diagram for the `command_executor` package is shown in Figure 4.6.

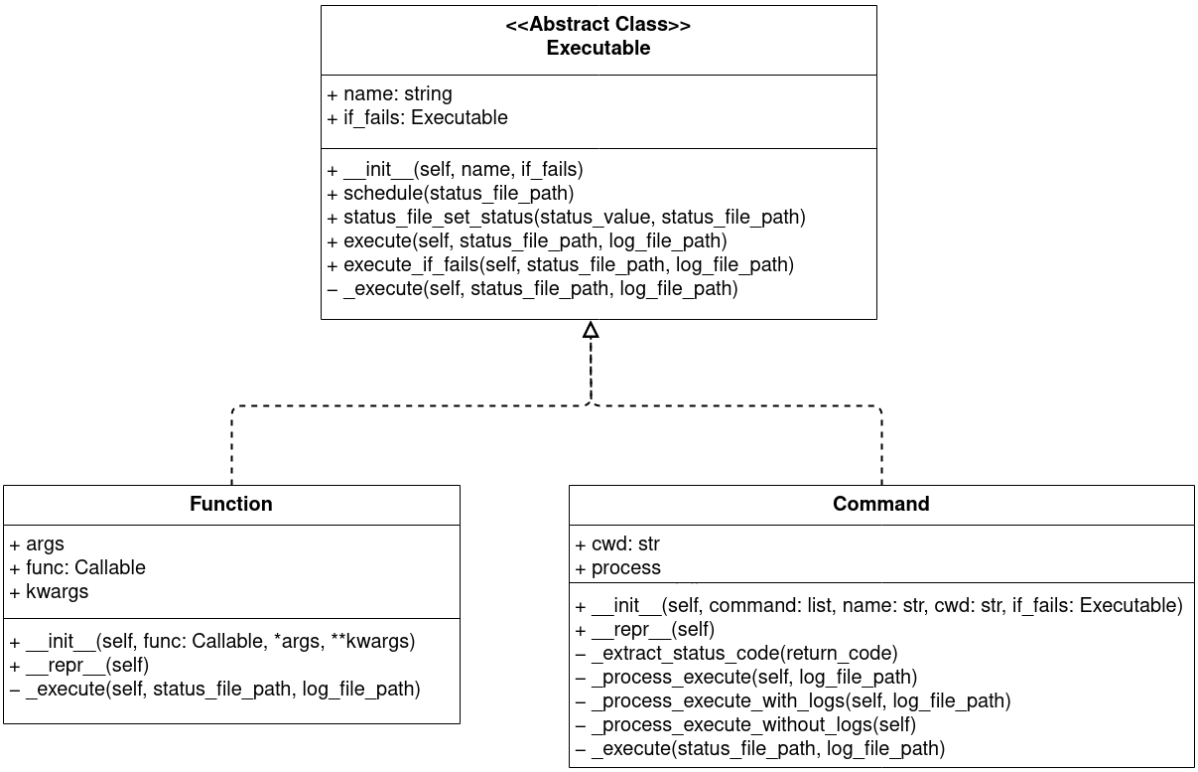


Figure 4.6.: Illustration of the classes of the `command_executor` package. Note that the `Executable` class provides implementations for every method except the abstract `_execute` method. The `if_fails` attribute is an Executable that is executed in case the main execution fails. This was implemented as sometimes the OpenMVS pipeline crashed due to errors with the CUDA integration.

4.6.6. Database Operations

Database operations are conducted with help of the `crud` package. This is again subdivided into the `GenericCRUD`, the `ProjectCRUD`, and the `TaskCRUD` classes. The `GenericCRUD` class provides the basic database operation functionalities such as creating, updating, querying, and deleting database instances. The `ProjectCRUD` and `TaskCRUD` provide more specific functions and shortcuts, as for instance getting a task or project by id. As the exact implementation of these more specific functions is unknown to the caller, a potential swap of databases would only require changes in the `crud` package. This extra layer of abstraction increases the maintainability and modularity of the system and reduces duplicate code to a minimum. An overview of the provided classes of the `crud` package can be seen in Figure 4.7.

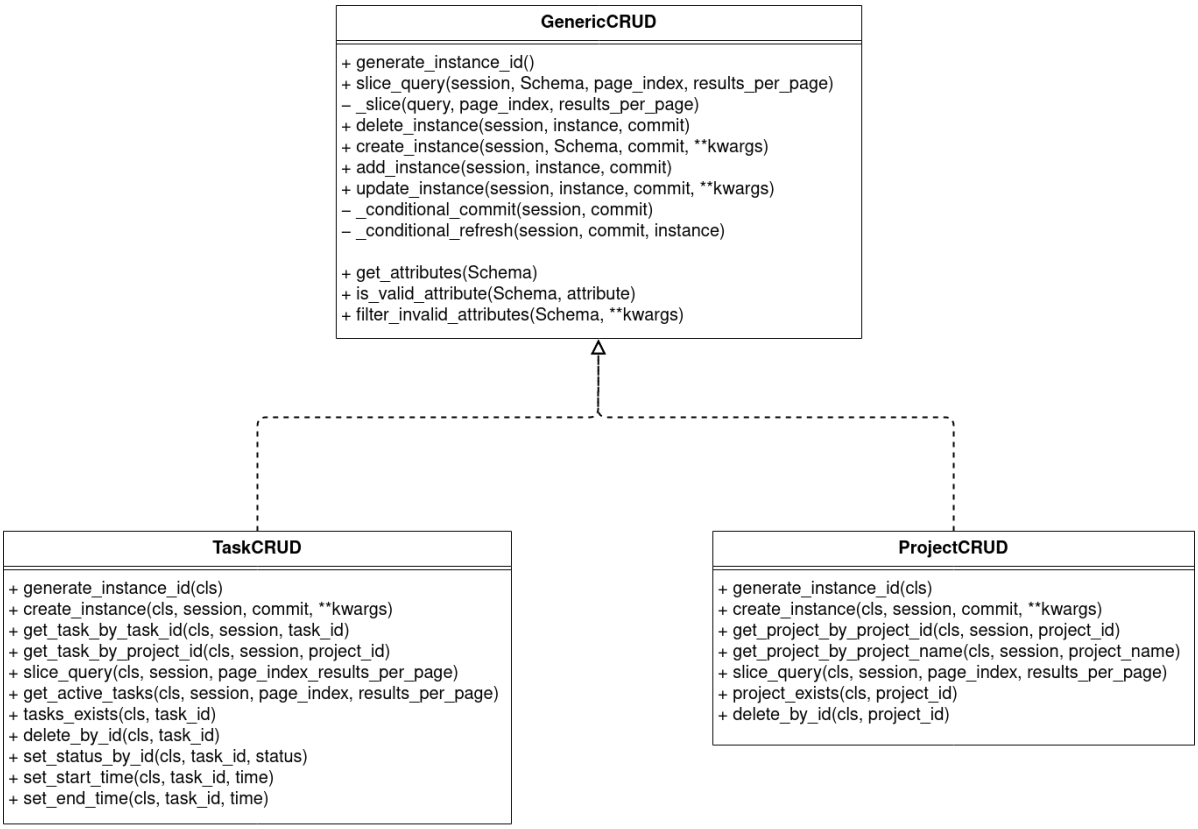


Figure 4.7.: Illustration of the classes of the `crud` package. The main functionalities are given by the `GenericCRUD` class, which are partially overridden by the child classes. The overridden methods then use the implementations provided by the parent class.

4.6.7. Server Configuration

The server's main configuration is done using the `config.py` files found in the individual service folders and the docker environment variables. The config files use the pydantic `BaseSettings` implementation, which allows it to be natively overridden by the environment variable of the same name. This allows them to be configured using our docker environment variables. The environment variables are specified in the `docker-compose.yml` file. The main use case for this is currently the configuration of service ports, number of worker threads (for our web microservices), application credentials, and some other settings.

4.6.8. Testing

We use some unit-tests for the `Backend-Manager-Service` to check the core functionalities of the service. These tests are focused on the metadata management aspect, asserting that all related file operations and API-routes work properly. Unit-tests for other components, besides minor tests for the `camera_poses` package, have not yet been implemented. During development, the Backend Manager Service was responsible for most of the system's dysfunction which is why we particularly integrated unit-tests here. Our other web-based applications are not very complex and are therefore not prone to many errors.

4.7. Provided Pipelines

In this section, we briefly go over the implemented photogrammetry pipelines.

Provider Meshroom Meshroom was the first pipeline we integrated into our system, as it provides a complete photogrammetry pipeline in one command-line command. Internally, it produces status files and log file information that we could use to extract task status information. This is one of the main reasons each provider currently has its own `/api/tasks/{task-id}/status` integration. The entire pipeline is structured as seen in Figure 4.8.

Provider COLMAP The second provider we integrated was COLMAP. Its pipeline is composed of multiple chained command-line calls, therefore requiring the implementation of the aforementioned `command_executor` package, which we also use as the main way of extracting status information. We offer two different pipelines: The `ColmapPipeline`, purely consisting of COLMAP commands, and the `ColmapOpenMVSPipeline`, where the dense reconstruction was replaced by OpenMVS commands. Both pipelines can be parameterized with the option `use_camera_poses`. If enabled, the pipeline uses the given camera pose information inside its reconstruction, meaning that it will skip the SfM step that is usually used to create the sparse model. The pipeline structure can be seen in Figure 4.9.

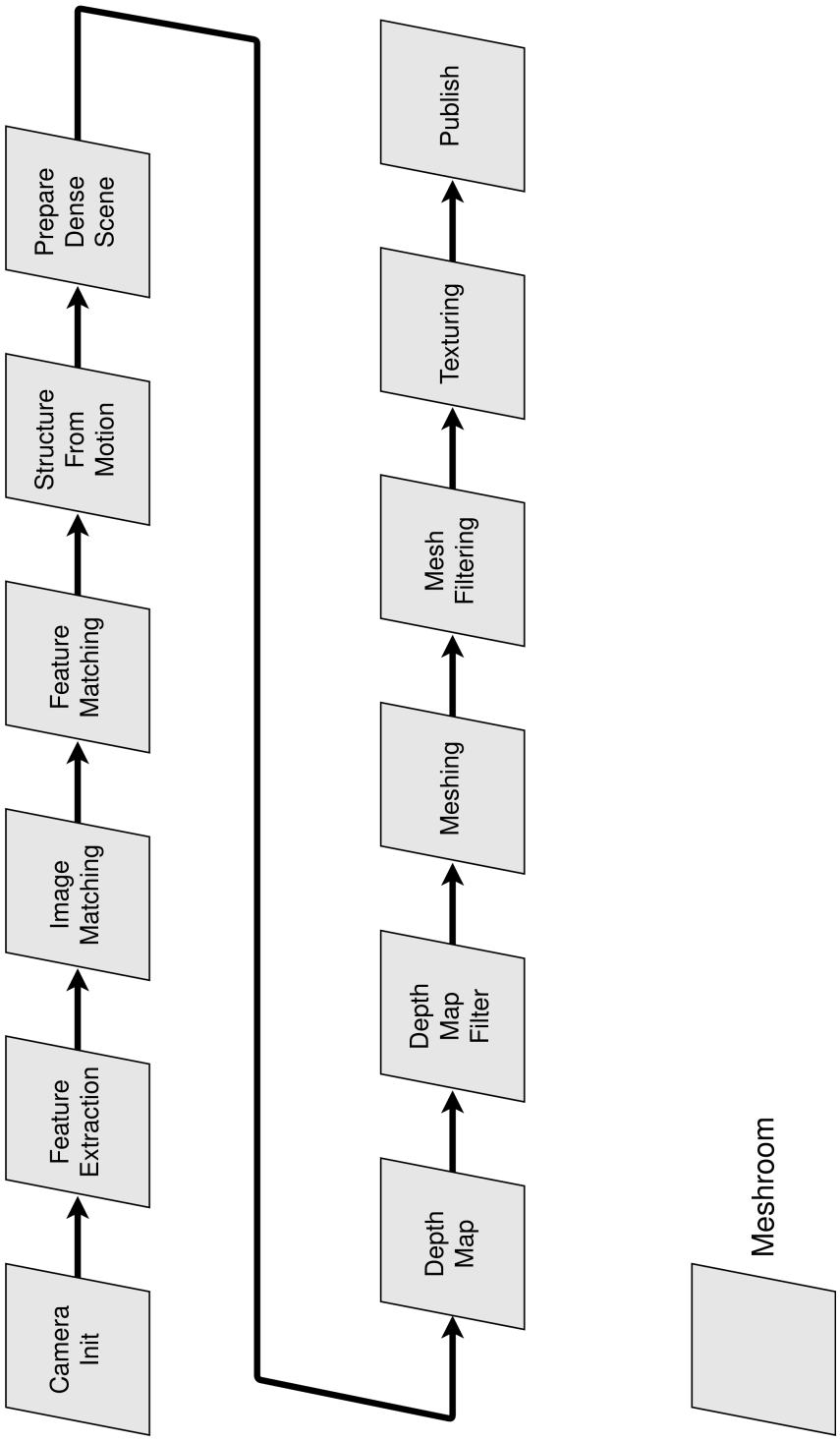


Figure 4.8.: Illustration of the photogrammetry pipeline for provider Meshroom. For more detailed information about the single steps, we refer to [Con20].

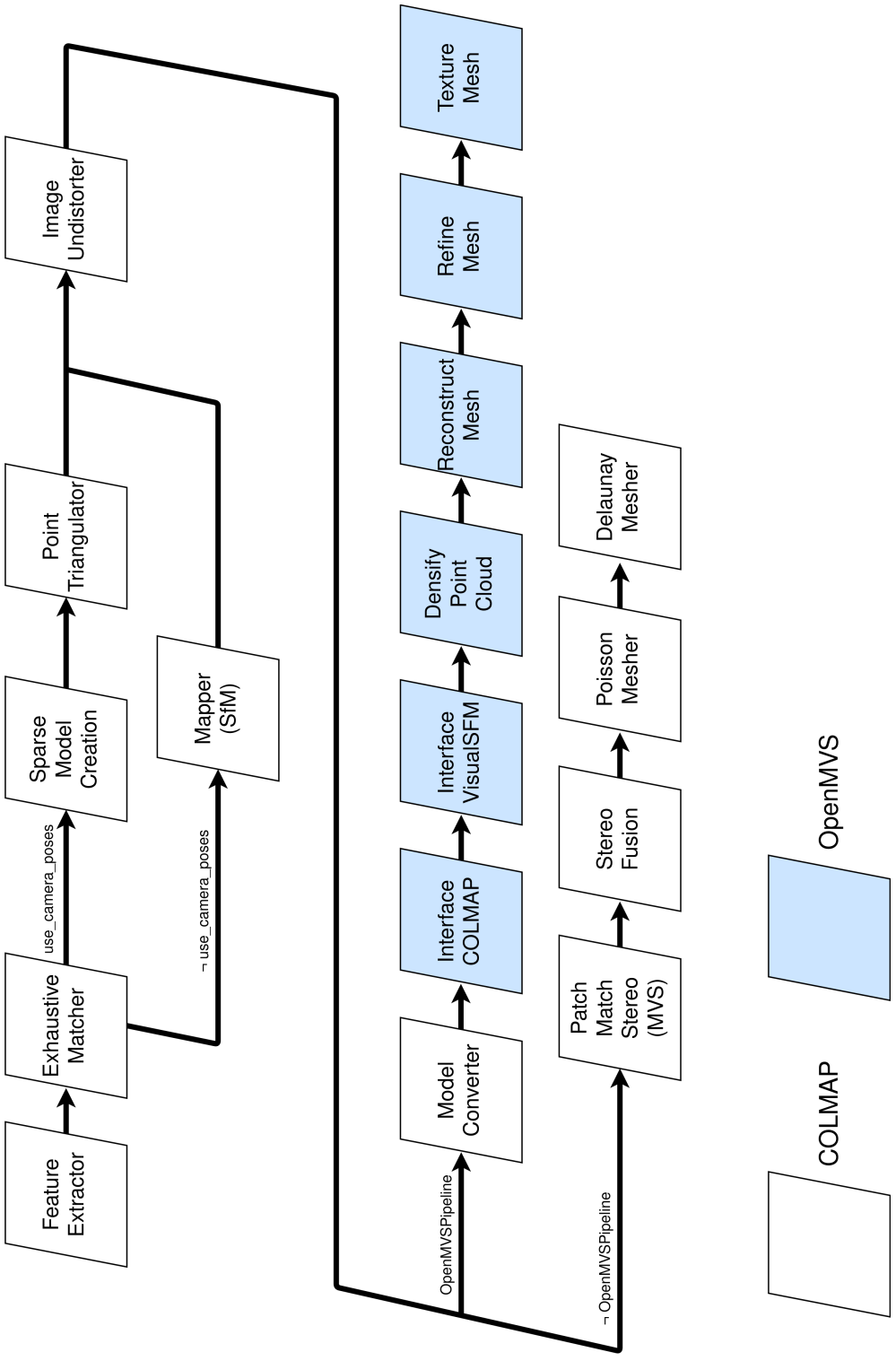


Figure 4.9.: Illustration of all available pipelines for provider COLMAP. For more detailed information about the used commands, we refer to [Schb] and [Cer] respectively.

5. Evaluation

In the next step, we want to evaluate our web service. We split our evaluation approach into the structural evaluation and the functional evaluation of our web service.

5.1. Structural Evaluation

The structural evaluation of our web service focuses on the client-side integration and the reusability aspect of our web service. We show how client applications can be integrated using our web service and give an overview of an example API workflow. Further, we examine whether each defined characteristic of a reusable web service could be satisfied.

5.1.1. Client Integration

In 2021, Stolz investigated the impacts of gamification-based user guidance on photogrammic 3D reconstruction. He proposed an AR-based application to guide users through the image acquisition step of the photogrammetry pipeline. In order to capture all necessary angles, a three-phase system is introduced directing the user around the object, taking pictures from different predefined viewpoints. Further gamification elements were added to keep users motivated. The reconstructions were conducted by hand using the photogrammetry software COLMAP. [Sto21]

Based on the AR poses of said application, we implemented the camera pose integration into our system. During the image acquisition steps, the AR pose can also be extracted and used in our web service. In the following, we will show how our service can be used to act as a suitable backend for the mentioned application.

Project Creation We can use a POST request to the `/api/projects/` route to create a project at the beginning of the image acquisition process.

Image Upload In the next step images can be uploaded using a POST request to the `/api/projects/{project_id}/images` route after each viewpoint in the gamification process. As we know the AR poses we can add this information as well.

Get Provider Descriptions If not known already, we can use a GET request on the `/api/providers/` route to get a list of all available task-descriptions and display it to the user.

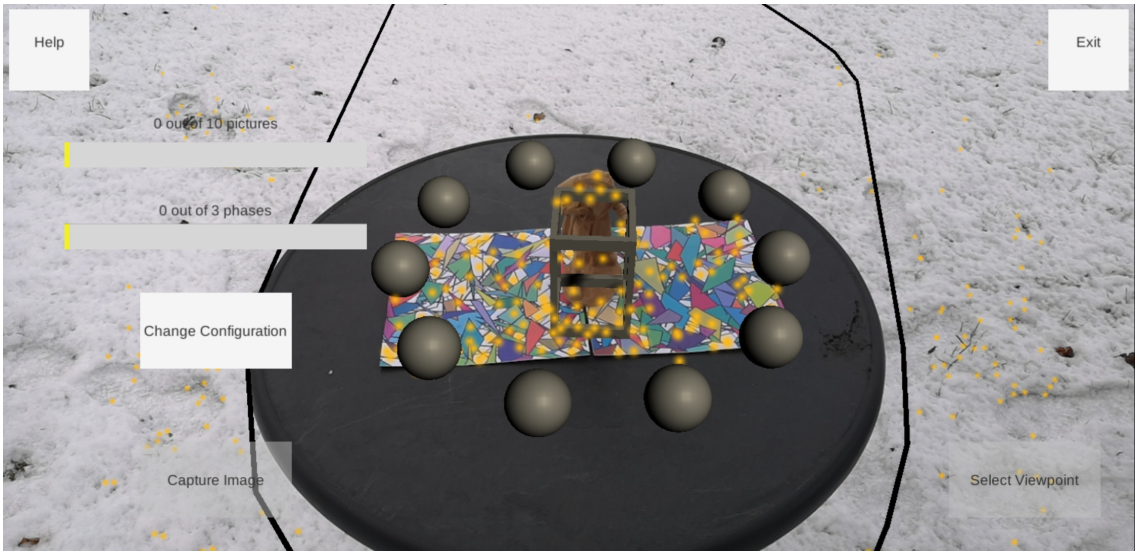


Figure 5.1.: Illustration of the gamification process. The spheres indicate the viewpoints from which the user needs to take pictures. Image taken from [Sto21].

Start Task We start the reconstruction task using a POST request to the `/api/tasks/` route with the user-chosen task-description and provider. When using the COLMAP provider we can further decide whether to use the implemented camera pose integration through the task parameters defined in the task-description.

Query Task Status Now we can query the task status through a GET request on the `/api/tasks/{task_id}/status` route and output the status to the user.

Download Files After task execution is finished we can download the output.zip, containing all output files, via GET request on `/api/tasks/{task_id}/output/output.zip` .

Using these routes we can utilize our web service in the client application making our web service a valid backend for the application.

5.1.2. Web Service Reusability

In the beginning of this thesis, we defined our requirements for a reusable web service. In this section, we will revisit each characteristic and evaluate whether or not our service complies with our definition of reusability.

Generalized Functionalities

Our web service provides a complete set of functionalities to manage image-sets, images, and image metadata through provided routes for upload, update, and deletion. Our web-API allows for a straightforward execution of tasks of a given project. As the task-description is

optional it can be used in a wide variety of ways. Clients seeking a basic reconstruction can use the API without the specification of further a task-description or parameters, whereas specialized clients can refine the reconstruction parameters using the task description. To sum up, it can be said that our web service does indeed provide generalized functionalities.

Documentation

As mentioned before, we provide our API documentation in form of the built-in Swagger integration. A detailed description of the required parameters, returned status codes and a textual description of the functionality of the API routes are given, usable for both client applications and future development. Hence, we also satisfy the documentation aspect of web service reusability.

Extensibility

We have already extended our web service based on new requirements that arose during the development of this system. The first expansion was the integration of the COLMAP provider. After we integrated the first provider Meshroom, it was found that other, newer alternatives exist that can be parameterized more easily than the existing implementation with Meshroom. As we already had an existing provider integration, the code could be reused for the new provider COLMAP. The integration worked flawlessly, as the pipeline could be easily constructed using the provided Command Executor package. The Backend Manager Service required little to no adjusting as the request dispatching process is almost fully dynamic and only needs to be parameterized using few variables.

The next feature expansion was the camera pose integration, motivated by the possible use of AR poses for the reconstruction. The web-API could be extended easily, as the project management is also dynamic and bound to a few key parameters. The new pipelines could be integrated into our system using updated task-description. The parameters for the camera pose were added to the task-description and the internal pipelines could be adjusted, again utilizing the Command Executor package. As it was discovered that the camera pose integration also requires the use of camera intrinsics, a new metadata field was added to the image model. This could also be implemented without effort. We also added the CPU images and depth images to the image metadata as they were part of new requirements set by client applications during the development of the system. As a result of the very dynamic parameterization of the Backend Manager Service, the addition of new image metadata information was very straightforward.

We can thus retain that our web service offers good extensibility.

Maintainability

We utilized a microservice in combination with containerization technology. As discussed earlier this has crucial benefits for maintainability.

As there is no objective measure of maintainability, the subsequent impressions are solely based on the author's opinion. The structure of the code was also designed in a modular fashion decomposing the code into intuitive components as remarked earlier. During the development of the web service, the code refactoring processes were straightforward as the responsibilities of each code component are defined clearly.

On the basis of the made structural decisions and applied code principles, the web service appears to be well maintainable.

5.2. Functional Evaluation

For our functional evaluation, we want to determine if the core functionalities of our systems work properly and test the stability of our system. Furthermore, we want to validate that our web-API can be used for our specified purposes. To achieve that, we test our application using different image-sets. We verify the functionality of each implemented pipeline, including our camera pose integration. At this point, it ought to be remarked that this is less focused on the evaluation of the providers, but more on the qualitative evaluation of the web service in general.

5.2.1. General Procedure

We test the functionalities using an evaluation script that accesses our web-API. We perform reconstructions on the image sets with every dataset with different providers and task-descriptions. For each dataset we test every available photogrammetry pipeline in our system:

- Provider Meshroom: MeshroomBatch pipeline
- Provider COLMAP: ColmapOpenMVSPipeline
- Provider COLMAP: ColmapPipeline

If the camera poses are available in the dataset, we will conduct one run with and one without the camera pose integration (for the provider COLMAP).

5.2.2. Evaluation Script

The evaluation script is found under `evaluation/scripts/evaluation_script.py` of our project's repository and works as follows:

1. We create a project using the corresponding API-route.
2. We add all images using HTTP-POST requests.
3. We start the task with the specified provider and task description. This is done in the form of a variable in the evaluation script file.

4. During the execution of the task we enter an infinite loop and query the task's status using the analogous route. We check our tasks status each minute and exit the loop as soon as the task is finished.

5. We use the output route to download the task's compressed `output.zip`.

6. Lastly, we query the task details to determine the start- and end-time.

7. We print the task duration and exit the script.

5.2.3. Evaluation Image-Sets

We use three different image sets: The LINEMOD dataset introduced by Hinterstoisser et al., Tanks and Temples 'Temple' dataset, presented by Knapitsch et al., and an own created dataset 'Bird Box'.

Bird Box Dataset This self-made dataset consists of 60 high-resolution images of a well-textured bird box. Example images can be observed in Figure 5.2.

Figure 5.2.: Example images of the 'Bird Box' dataset.

LINEMOD Dataset To test our camera pose integration we utilize the LINEMOD dataset[1]. We use a subset of the LINEMOD dataset consisting of 1254 images. It shall be noted that the main purpose of this dataset is not for 3D reconstruction, but for 6D pose estimation. The image resolution is quite low, which is not ideal for photogrammetry applications. However, as we currently do not have another dataset with ground-truth poses and intrinsics available, we will make use of the LINEMOD dataset. Figure 5.3 shows some example images of the image set.

[1] https://bop.felk.cvut.cz/datasets/

Figure 5.3.: Example images of the LINEMOD dataset [Hin+12].

Temple Dataset The 'Temple' dataset[2] was specifically created for 3D reconstruction benchmarking. It consists of 302 high-quality images all around the building. Example pictures are shown in Fig 5.4.

Figure 5.4.: Example images of the 'Temple' dataset [Kna+17].

5.2.4. Pipeline Functionality

We conducted runs for each dataset for each available pipeline configuration. All tasks were executed and finished successfully with the exception of the 'Temples' dataset. Testing our COLMAP pipeline on the 'Temples' dataset resulted in an error. This was due to the lack of sufficient RAM on the testing system which caused the reconstruction process to be killed by the OS. Other internal errors were not observed during the testing so we can conclude the basic functionalities work as expected. The interested reader can find the resulting models on the GitLab repository [Krü21].

5.2.5. Camera Pose Integration

The camera pose integration was tested using the COLMAP provider and the mentioned LINEMOD dataset. The task execution was successful each time and we could retrieve the corresponding output model. In the following, we will briefly compare the model obtained from the ColmapOpenMVSPipeline with and without the camera pose integration to briefly demonstrate how our web service helps to compare photogrammetry pipelines.

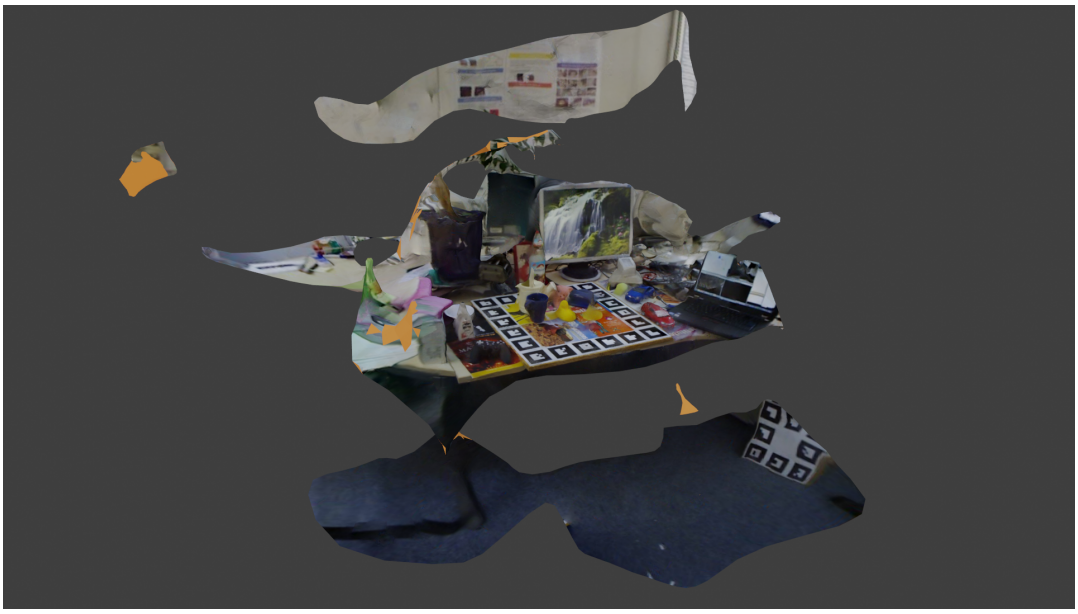
The two retrieved meshes can be seen in Figure 5.5. Both meshes have roughly the same quality, but the model without the camera pose integration could pick up more of the background. The textures of the model using the camera pose integration are a touch sharper. In general, there is no big difference between both models. In terms of time the model

[2]https://www.tanksandtemples.org/download/

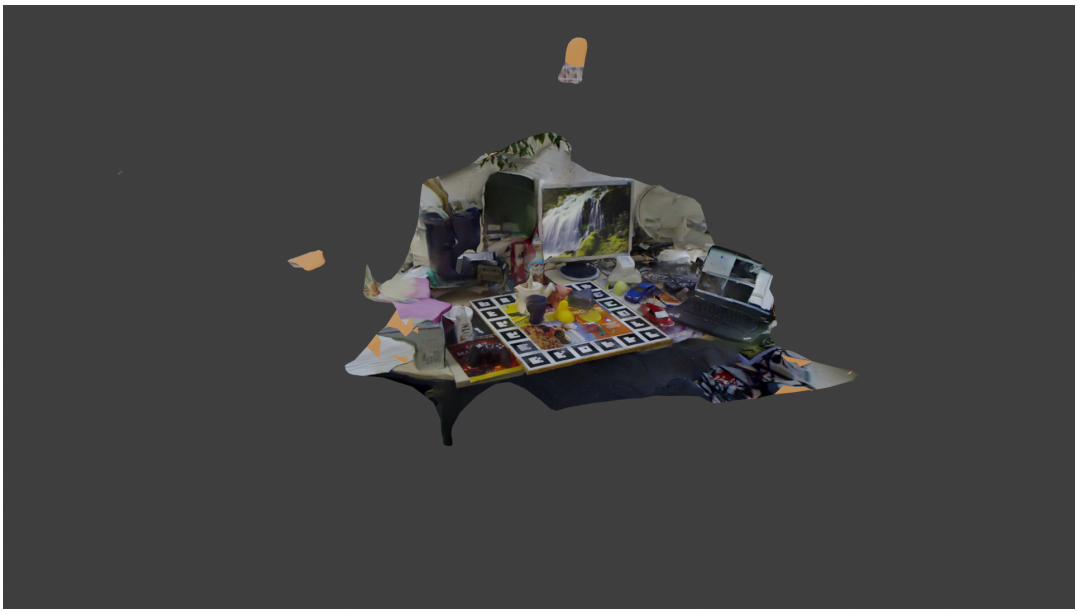
utilizing the camera pose integration was roughly 30 minutes faster, taking only one hour in total.

5.3. Summary

In this section we have shown our system meets the required specifications and can be used by client applications as a valid backend application. The task execution has been without error during our testing, except for errors caused by RAM limitations, showing the stability of our system. Furthermore, we have seen that our web service allows us to compare models created through different pipelines and configurations which was one of the initial goals of this thesis.



(a) Model produced without the camera pose integration.



(b) Model produced with the camera pose integration.

Figure 5.5.: Comparison between models created with and without the camera pose integration. Note that (a) could recover more of the background of the scene.

6. Discussion

We have shown that our system works as expected and our structure satisfies our defined requirements, however, like in every system under construction, there is always room for improvement. In this chapter, we discuss lacking or suboptimally implemented aspects of our web service. Note that these impressions are based on the author's opinion.

As remarked earlier, the general functionalities work as expected for most of the time, though errors can not be excluded. Currently, we have not implemented extensive error handling in the web service, for example when individual components dysfunction or become unavailable. This has not been a big issue during the development as there were only very few exceptions occurring that were not related to the general functionality of the system. In the worst case, the service restarted after an uncaught internal exception. Nonetheless, for the completeness and stability of the web service an extensive error handling system could have been implemented.

Another closely related aspect is the lack of logging. It was attempted to add this functionality during the development as it is a key factor in the monitoring and debugging of the application. Python offers an easy way to implement logging into any system, but during the implementation of said logging functionality, it was discovered that the deployment with Gunicorn and Uvicorn makes the initially easy logging a lot more complex. Due to the lack of time and comparably high effort involved in the implementation of logging into our web service, it was not implemented into the final system.

Meshroom was the first provider we integrated into the system. It was chosen as it offers a complete photogrammetry pipeline in a single command-line command. Initially, this allowed for very easy integration of the provider. During later stages of development, with the implementation of the task-description allowing for a precise definition of task parameters, it was discovered that the Meshroom's parameterization options are not ideal for the integration into an automated pipeline. Its graph-based structure is great for the use in graphical user interfaces, but using the AliceVision framework directly might be a better choice for the integration into our system as we could utilize the individual command-line commands directly. Swapping Meshroom with AliceVision was considered during the development but not implemented as the integration of the COLMAP provider was prioritized higher.

Another minor flaw in the implementation was the integration of the CPU image. In later stages, it was discovered that the use of the CPU image as image metadata is redundant as its main use case would have been the use of CPU images as the basis of reconstruction which can equally be done using the image field of the image model.

The made structural decisions that were mentioned in this thesis provided the desired benefits and there were no negative aspects of the made decisions noticed. This further implicates that the right decisions were made in regards to the proposed system architecture.

7. Conclusion

7.1. Summary

In this thesis, we proposed a well-functioning and reusable web service for online 3D reconstruction providing web-API with generalized functionalities for client applications. Our architecture uses state-of-the-art technologies with the use of containerization technology and the microservice architecture. Moreover, our web service was developed in a future-oriented manner. It provides basic and advanced functionalities for expert and non-expert users and a well-documented API. Further, we implemented three different photogrammetry software into two different providers and have shown how our service can be used to upload images to image-sets, start reconstructions and compare different providers. The made structural and code design decisions have made the service extensible and reusable. We have evaluated the structure and functionality of our system showing that our web service works as intended and satisfies our reusability requirements. Though there are some minor things that could not at all or not sufficiently be implemented in the scope of this thesis, we have laid the structural foundations for future development and further refinement of the provided features and server architecture.

7.2. Future Work

Integration of More Providers We have proposed a web service, that integrates photogrammetry software from AliceVision Meshroom, COLMAP, and OpenMVS. In the next step more providers should be implemented in order to make our web service more diverse and powerful.

Use of Additional Information We integrated the use of camera pose information into the provider COLMAP. The use of more additional information such as depth information through the use of depth images should be implemented and investigated in the future. The integration of depth information should contribute to a more realistic scaling of the resulting 3D model.

Use of Other Input Formats As of now, our system uses image data as the base of our reconstructions. More input types such as video input data could be implemented in future work projects.

Further Modularization of Pipeline Steps Our providers have the ability to execute different tasks with given parameters. For further research, one could investigate the possibility to break down these pipeline steps into the modular components of each pipeline. These new task types could be used to stitch together more complex pipelines using different photogrammetry software providing even more customization options to the client applications.

Reconstruction Metadata We have already added the structural foundations to add metadata to the reconstruction tasks. However, further implementations need to be done to provide more detailed information about the reconstruction task. Metrics and metadata, like number of images, number of successfully processed images, camera lens models, number of triangulated points, number of points for the sparse or dense point cloud and many more, as implemented by Tefera et al., could also be integrated into our system. This could provide valuable information and would help to compare different models made with other parameters or created using alternative providers.

Evaluation of the Models Further future work could include the integration of a model evaluation service. Adding a ground truth model to the project and comparing it with the task output could be a vital metric for comparability.

Parameter Tuning We have introduced a scheme that allows us to pass parameters to our providers. A next step in research could be finding algorithms to tune our reconstruction parameters by using an objective evaluation process. For that to happen, we need to implement more parameterization options into our task-descriptions.

Scaling With Kubernetes The proposed architecture is designed to be scaled in the future. The provided docker images and used technologies should make an integration into a container orchestration tool like Kubernetes feasible. This could help deploy the application into cloud computing platforms like the Google Cloud Platform (GCP), which makes scaling the services vertically, by adding more CPU and RAM resources, as well as horizontally, by adding more worker threads or machines, possible.

A. API Route Specifications

For the full API route specifications, we refer to the built-in Swagger documentation in our web service. Each web-based microservice in our system offers two endpoints for API documentation. The `/api/swagger` route provides the mentioned Swagger documentation, whereas the `/api/redoc` route redirects to the redoc[1] documentation. Given a local deployment the documentation can be accessed as seen in Table A.1. A small excerpt of the swagger documentation of the Backend Manager Service can be seen in Figure A.1.

Microservice	Swagger URL	Redoc URL
Backend Manager Service	`http://localhost:8011/api/swagger`	`http://localhost:8011/api/redoc`
COLMAP Service	`http://localhost:8002/api/swagger`	`http://localhost:8002/api/redoc`
Meshroom Service	`http://localhost:8001/api/swagger`	`http://localhost:8001/api/redoc`

Table A.1.: List of the available documentation routes

[1] https://github.com/Redocly/redoc

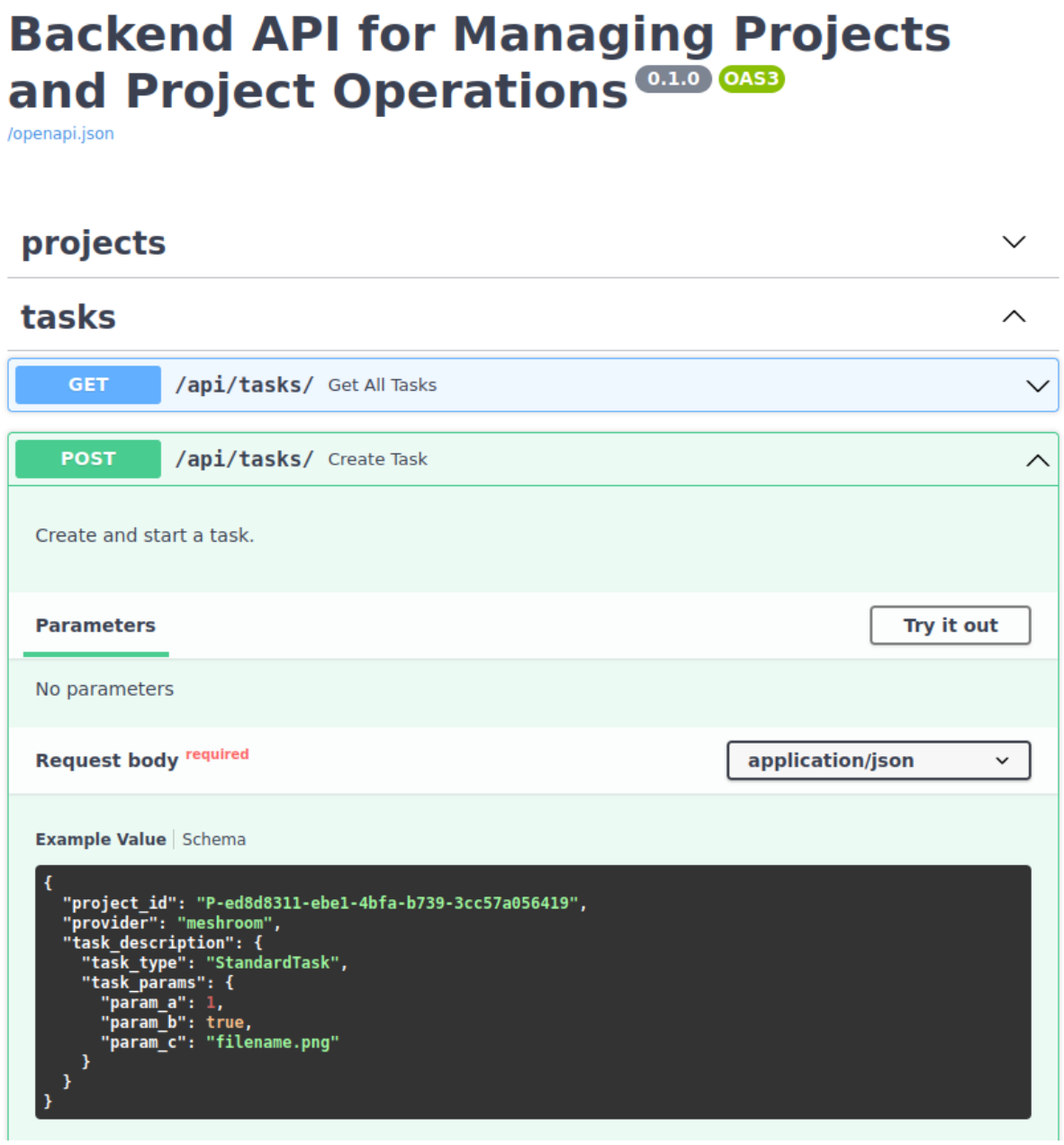


Figure A.1.: Screenshot of the Swagger documentation of the Backend Manager Service. The Swagger documentation provides detailed information about the required payload, returned response codes and enables the testing of routes via browser.

B. Pydantic Model

An example pydantic model is defined in Listing B.1. The pydantic also supports model nesting, which means that models can use other models within themselves.

```
from pydantic import BaseModel

class ProjectInitialization(BaseModel):
    project_name: str
    project_owner: Optional[str]
```

Listing B.1: Simplified code of our `ProjectInitialization` model, which is used during project creation. The project_name is marked as a required string, whereas the project_owner is marked optional. This model will automatically be validated by FastAPI. Invalid route payload will result in an HTTP response with the status code: 422 Unprocessable Entity and a description of the invalid fields.

List of Figures

List of Tables

Bibliography

[HLB19] X.-F. Han, H. Laga, and M. Bennamoun. "Image-based 3D object reconstruction: State-of-the-art and trends in the deep learning era". In: *IEEE transactions on pattern analysis and machine intelligence* 43.5 (2019), pp. 1578–1604.

[Sch05] T. Schenk. "Introduction to photogrammetry". In: *The Ohio State University, Columbus* 106 (2005).

[Lin09] W. Linder. *Digital photogrammetry*. Vol. 1. Springer, 2009.

[Lac17] J. C. Lachambre S Lagarde S. "Unity Photogrammetry Workflow". In: (2017). URL: https://unity3d.com/files/solutions/photogrammetry/Unity-Photogrammetry-Workflow_2017-07_v2.pdf (Last accessed: 2021-08-12).

[Gep+20] M. Geppert, V. Larsson, P. Speciale, J. L. Schönberger, and M. Pollefeys. "Privacy Preserving Structure-from-Motion". In: *European Conference on Computer Vision (ECCV)*. 2020.

[Aga+10] S. Agarwal, Y. Furukawa, N. Snavely, and I. Simon. 2010. URL: http://grail.cs.washington.edu/rome/dense.html (Last accessed: 2021-08-12).

[FHP15] Y. Furukawa, C. Hernández, and N. Publishers. *Multi-view Stereo: A Tutorial*. Foundations and trends in computer graphics and vision. Now Publishers, 2015. ISBN: 9781601988379. URL: https://www.nowpublishers.com/article/DownloadSummary/CGV-052 (Last accessed: 2021-08-12).

[Sac+20] E. Saczuk et al. *Processing Multi-spectral Imagery with Agisoft MetaShape Pro*. 2020. URL: https://pressbooks.bccampus.ca/ericsaczuk/chapter/chapter-2-1-dense-point-cloud/ (Last accessed: 2021-08-12).

[Bau] A. Baumberg. "Blending Images for Texturing 3D Models." In: Citeseer.

[Alia] AliceVision. *Meshroom*. URL: https://alicevision.org/#meshroom (Last accessed: 2021-08-12).

[Alib] AliceVision. *AliceVision*. URL: https://github.com/alicevision/AliceVision (Last accessed: 2021-08-12).

[Scha] J. L. Schönberger. *COLMAP*. URL: https://colmap.github.io/ (Last accessed: 2021-08-12).

[SF16] J. L. Schönberger and J.-M. Frahm. "Structure-from-Motion Revisited". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[Sch+16] J. L. Schönberger, E. Zheng, M. Pollefeys, and J.-M. Frahm. "Pixelwise View Selection for Unstructured Multi-View Stereo". In: *European Conference on Computer Vision (ECCV)*. 2016.

[Cer20] D. Cernea. "OpenMVS: Multi-View Stereo Reconstruction Library". 2020. URL: https://cdcseacave.github.io/openMVS (Last accessed: 2021-08-12).

[Ora21] Oracle. *Learn about architecting microservices-based applications on Oracle Cloud*. 2021. URL: https://docs.oracle.com/en/solutions/learn-architect-microservice/index.html#GUID-1A9ECC2B-F7E6-430F-8EDA-911712467953 (Last accessed: 2021-08-12).

[Kap20] S. Kappagantula. *What Is Microservices – Introduction To Microservice Architecture*. 2020. URL: https://www.edureka.co/blog/what-is-microservices/ (Last accessed: 2021-08-12).

[Edu21] I. C. Education. *Containerization*. 2021. URL: https://www.ibm.com/cloud/learn/containerization (Last accessed: 2021-08-12).

[Hat] R. Hat. *Containers vs VMs*. URL: https://www.redhat.com/en/topics/containers/containers-vs-vms (Last accessed: 2021-08-12).

[Han00] M. D. Hanson. "The Client/Server Architecture". In: *Server Management* (2000).

[Red11] M. Reddy. *API Design for C++*. Elsevier Science, 2011. Chap. 1. ISBN: 9780123850041. URL: https://books.google.de/books?id=IY29LylT85wC (Last accessed: 2021-08-12).

[SR19] H. Subramanian and P. Raj. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs*. Packt Publishing Ltd, 2019. Chap. REST architectural constraints. URL: https://learning.oreilly.com/library/view/hands-on-restful-api/9781788992664/ (Last accessed: 2021-08-12).

[Cas07] T. Cashion. *Rails Refactoring to Resources (Digital Short Cut): Using CRUD and REST in Your Rails Application*. Pearson Education, 2007. Chap. 2.1. URL: https://learning.oreilly.com/library/view/rails-refactoring-to/9780321501745/ (Last accessed: 2021-08-12).

[Hel+15] J. Heller, M. Havlena, M. Jancosek, A. Torii, and T. Pajdla. "3D reconstruction from photographs by CMP SfM web service". In: *2015 14th IAPR International Conference on Machine Vision Applications (MVA)*. IEEE. 2015, pp. 30–34.

[Tef+18] Y. T. Tefera, F. Poiesi, D. Morabito, F. Remondino, E. Nocerino, and P. Chippendale. "3DNOW: Image-based 3D reconstruction and modeling via web". In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* 42.2 (2018), pp. 1097–1103.

[Aut] Autodesk. *ReCap Pro*. URL: https://www.autodesk.com/products/recap/overview (Last accessed: 2021-08-12).

[LLC20] A. LLC. *Metashape Presentation*. 2020. URL: https://www.agisoft.com/pdf/metashape_presentation.pdf (Last accessed: 2021-08-12).

[TG14] S. Taneja and P. R. Gupta. "Python as a tool for web server application development". In: *JIMS8I-International Journal of Information Communication and Computing Technology* 2.1 (2014), pp. 77–83.

[Krü21] M. Krüger. *ba-ss21-krueger-moritz-photogrammetry-backend*. 2021. URL: https://gitlab.lrz.de/IN-FAR/Thesis-Projects/ba-ss21-krueger-moritz-photogrammetry-backend (Last accessed: 2021-08-12).

[Cel] Celery. *Backends and Brokers - Celery 5.1.2 documentation*. URL: https://docs.celeryproject.org/en/stable/getting-started/backends-and-brokers/index.html (Last accessed: 2021-08-12).

[Sze10] R. Szeliski. *Computer vision: algorithms and applications*. Springer Science & Business Media, 2010. Chap. 2.1 Geometric primitives and tranformations.

[Con20] M. Contributors. *Node Reference - Meshroom 19.02.003 documentation*. 2020. URL: https://meshroom-manual.readthedocs.io/en/bibtex1/node-reference/node-reference.html (Last accessed: 2021-08-12).

[Schb] J. L. Schönberger. *Command-line Interface - COLMAP 3.7 documentation*. URL: https://colmap.github.io/cli.html (Last accessed: 2021-08-12).

[Cer] D. Cernea. *OpenMVS Github*. URL: https://github.com/cdcseacave/openMVS/tree/master/apps (Last accessed: 2021-08-12).

[Sto21] S. Stolz. *Improving photogrammetric 3D Reconstruction by Augmented Reality and Gamification based User Guidance*. 2021.

[Hin+12] S. Hinterstoisser, V. Lepetit, S. Ilic, S. Holzer, G. Bradski, K. Konolige, and N. Navab. "Model based training, detection and pose estimation of texture-less 3d objects in heavily cluttered scenes". In: *Asian conference on computer vision*. Springer. 2012, pp. 548–562.

[Kna+17] A. Knapitsch, J. Park, Q.-Y. Zhou, and V. Koltun. "Tanks and Temples: Benchmarking Large-Scale Scene Reconstruction". In: *ACM Transactions on Graphics* 36.4 (2017).