10.01.2025 - Bachelor thesis fall semester 2024/25

BoltFinder

A Prototype for ML-Assisted Climbing Route Digitization

Students: **Nick Wallner** **Fabio Elvedi**

Supervisor: **Prof. Dr. Mitra Purandare**

Co-Supervisor: **Lars Herrmann**

Industry-partner: **Marek Polacek**
SAC: Schweizer Alpen-Club

Abstract

Switzerland has around 47,000 climbing routes, yet detailed information about these routes is fragmented across various platforms and printed guidebooks. The Swiss Alpine Club (SAC), a leading climbing organization, currently manages climbing sectors manually, highlighting the lack for automation. An automated solution poses challenges, such as gathering a comprehensive dataset, detecting small objects such as bolts and anchors, and using algorithmic traversal of predictions to determine accurate climbing routes for diverse areas. Tackling these challenges streamlines climbing route digitization.

BoltFinder is a proof-ofconcept solution leveraging machine learning to automate digitization of climbing routes. BoltFinder uses YOLOv11 to detect bolts and anchors in climbing wall images. Based on these detections, routes are identified through an algorithm that leverages Voronoi diagrams. A React-based frontend offers a user-friendly interface for reviewing and editing predictions and routes. The backend consists of an ExpressJS API for managing the business workflow, including the route generation algorithm, and a FastAPI responsible for handling image tiling and generating predictions. The PostgreSQL database employs a relational data model to organize and manage route structures. Model tracking is managed through MLFlow, ensuring robust performance monitoring, scalability, and adherence to MLOps principles.

The prototype uses YOLOv11 to detect bolts and anchors. Using 1024px tiles, the model exhibits a mAP@25 score of 0.89 for bolts. YOLOv11 surpasses Faster R-CNN in precision and F1 score by 20%. The system identifies climbing routes with an accuracy of 78% across 102 routes. 21 identified routes exhibit minor deviation whereas major deviation is observed in only one route. These results establish BoltFinder as an automated solution for climbing route digitization, providing significant value to climbers and the SAC. The prototype shows strong potential. Further improvements are possible via drone imagery, classifying different bolt types, and identifying bolt conditions like rust or wear.

Acknowledgements

We extend our heartfelt gratitude to those who contributed to the success of this project.

We are deeply thankful to Prof. Dr. Mitra Purandare for her invaluable guidance and unwavering support throughout the development of this work. Her insights and encouragement were instrumental in navigating challenges and achieving our objectives.

Our sincere thanks go to Lars Herrmann for his support and thoughtful advice, which provided clarity and direction during critical phases of the project.

We are also grateful to Marek Polacek for initiating this project and sharing his expertise. His constructive feedback and insights significantly enriched the quality of our work.

Special thanks to Prof. Stefan Keller for his advice and support, which added depth and perspective to the project.

Finally, we appreciate Datong Xu for his guidance in identifying and refining the algorithm for route detection, a pivotal aspect of this research.

To all who offered their time, expertise, and encouragement, we express our deepest appreciation. Your contributions were integral to the success of this project.

1 Management Summary

Starting Situation

Climbing in Switzerland boasts an impressive array of routes, with approximately 47,000 routes documented across thousands of climbing areas. However, information about these routes—such as grading, safety, rope length, and condition—remains fragmented across various online platforms and printed guidebooks. The Swiss Alpine Club (SAC), the largest climbing organization in the country, holds a wealth of data but currently has to maintain the data manually, there is a need for an automated solution to digitize climbing routes.

Task Definition

This bachelor's thesis tackles the challenge by developing a proof-of-concept (POC) solution for automated digitization of climbing routes. Leveraging machine learning, the project introduces a novel approach to detect climbing infrastructure, like bolts and rope anchors, from images of outdoor climbing walls. The detected infrastructure enables route identification and visualization. The system integrates a backend for image processing, a frontend for manual data refinement, and visualization capabilities to produce topographic representations (topo).

Analysis

Digitization of climbing routes in its current state relies heavily on manual efforts. Printed guidebooks and online platforms, such as TheCrag, Kletterfux, and Vertical Life, serve as primary sources of information. However, these methods lead to fragmented data, with varying formats and inconsistencies. The reliance on manual or community-based data input further complicates the standardization and accessibility of climbing route information.

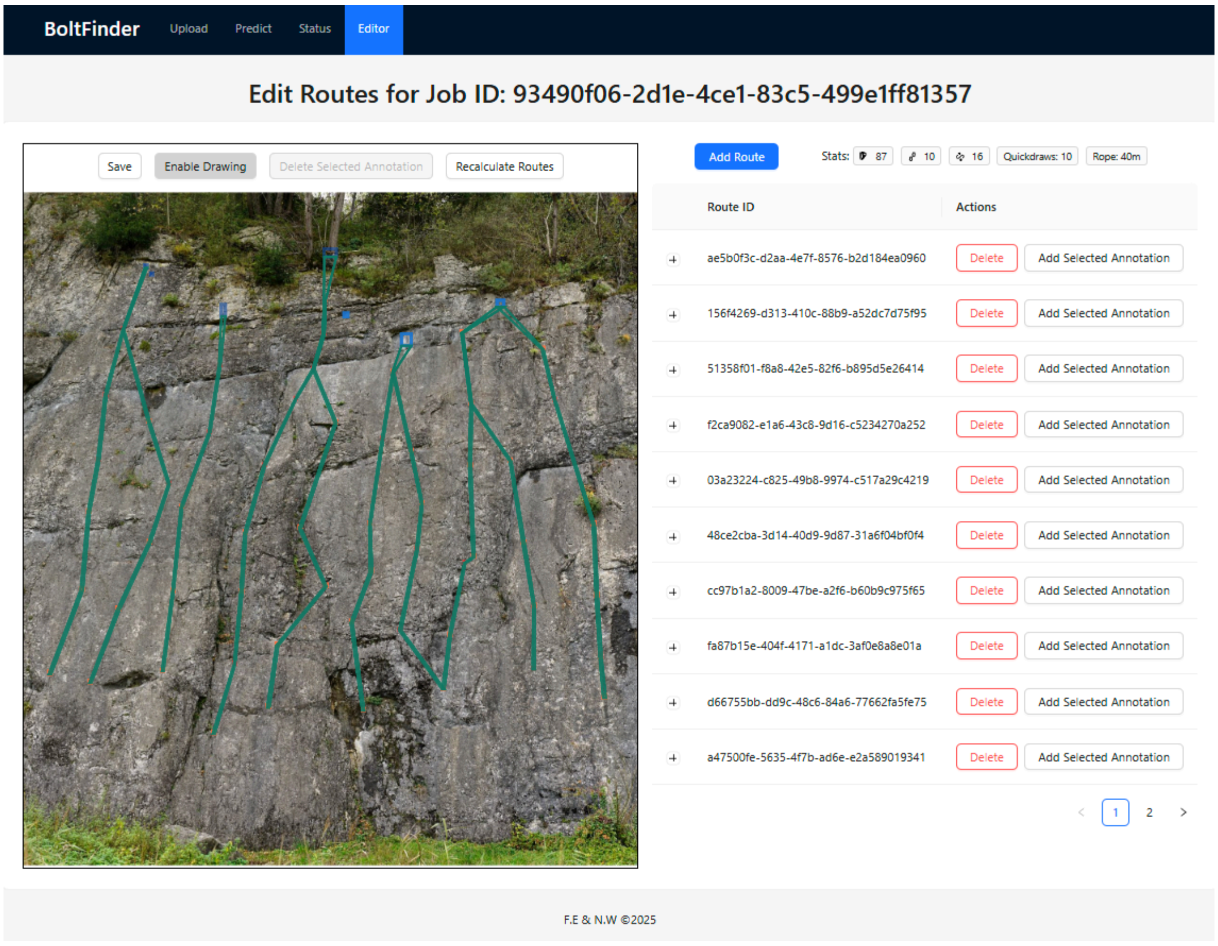
The challenges associated with the documentation of climbing routes are particularly pronounced in remote regions, where access and maintenance can be challenging. Furthermore, the distinct nature of each rock climbing sector necessitates customized topographic representations, thereby introducing an additional layer of complexity to the process.

From a requirements perspective, the SAC aims to evaluate whether a solution combining a machine learning model for detecting climbing infrastructure from images and an algorithm for identifying climbing routes can effectively address their needs. They also want to see if this approach can be used in many climbing areas. For climbers, the most important thing is having easy access to accurate route information and topos. These tools help climbers find and plan their favorite routes.

This analysis underscores the need for an innovative solution that bridges these gaps, offering both climbers and organizations like the SAC a unified, efficient approach to documentation of climbing routes.

Realization

The implementation employs innovative technologies, including YOLOv11 for object detection, Express.js for backend development, and React for creating a user-friendly interface. Tools like MLFlow are utilized for model tracking, ensuring a comprehensive technical foundation for the proof-of-concept. This robust infrastructure positions the system for potential extension into real-world applications. The project's technical component can be interacted with the BoltFinder Website, which provides a user-friendly interface for uploading images, viewing predictions, as well as editing climbing routes as shown in the Figure 1 below.



Edit Routes for Job ID: 93490f06-2d1e-4ce1-83c5-499e1ff81357

	Route ID	Actions
+	ae5b0f3c-d2aa-4e7f-8576-b2d184ea0960	Delete / Add Selected Annotation
+	156f4269-d313-410c-88b9-a52dc7d75f95	Delete / Add Selected Annotation
+	51358f01-f8a8-42e5-82f6-b895d5e26414	Delete / Add Selected Annotation
+	f2ca9082-e1a6-43c8-9d16-c5234270a252	Delete / Add Selected Annotation
+	03a23224-c825-49b8-9974-c517a29c4219	Delete / Add Selected Annotation
+	48ce2cba-3d14-40d9-9d87-31a6f04bf0f4	Delete / Add Selected Annotation
+	cc97b1a2-8009-47be-a2f6-b60b9c975f65	Delete / Add Selected Annotation
+	fa87b15e-404f-4171-a1dc-3af0e8a8e01a	Delete / Add Selected Annotation
+	d66755bb-dd9c-48c6-84a6-77662fa5fe75	Delete / Add Selected Annotation
+	a47500fe-5635-4f7b-ad6e-e2a589019341	Delete / Add Selected Annotation

< 1 2 >

Figure 1: BoltFinder Editing UI

Results

The BoltFinder prototype demonstrates strong capabilities in detecting climbing bolts and anchors in images, enabling automated identification and visualization of climbing routes. Through extensive testing, the algorithm evaluated a total of 102 routes, with eighty routes correctly identified, achieving an accuracy of 78%. Minor deviations are observed in twenty-one routes, and only a single route exhibited major errors, highlighting the system's reliability in most cases. The final topographic representation and the Voronoi representation is shown in the Figure 2 below.

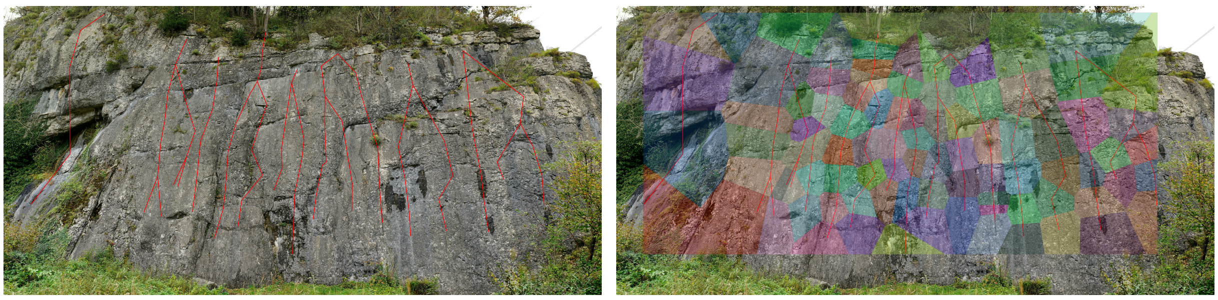


Figure 2: Left: Topo Representation of Climbing Routes, Right: Voronoi Diagram of Climbing Routes

Two types of machine learning models, Faster R-CNN and YOLOv11, are trained and tested. The YOLOv11 model, trained on all available data (combined model), emerged as the best-performing solution. It successfully predicts approximately 90% of bolts with only 20% false positives. The model is also tested with varying tile sizes, ranging from 512px to 1024px, and the 1024px tile size proved to be the most practical. YOLOv11 achieved significantly better F1 scores, particularly for anchor detection, and demonstrated higher precision compared to the Faster R-CNN model, reducing false positives and improving the system's overall usability.

The following Figure 3 illustrates the predictions made by the combined YOLOv11 model, showcasing its ability to detect and annotate bolts and anchors. Green bounding boxes indicate ground truth, while red bounding boxes represent the model's predictions.

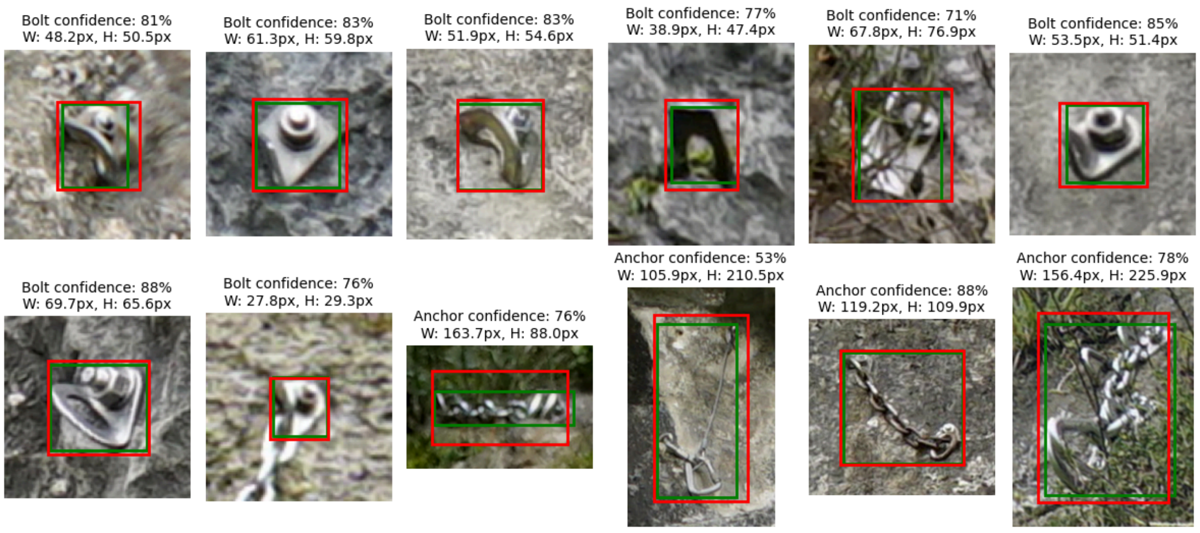


Figure 3: Predictions from the YOLOv11 Combined Model

These results validate the BoltFinder system as an effective and scalable solution for climbing route documentation. This prototype provides climbers and organizations, such as the SAC, with a reliable and efficient tool for managing climbing route data.

Outlook

The project opens the door to numerous possibilities for extending its scope and functionality. Potential future enhancements include integrating additional metadata, such as rock type or bolt condition, enhancing route difficulty predictions, and addressing more complex climbing scenarios, such as multi-pitch routes. These enhancements could significantly broaden the applicability and utility of the system.

Drone-captured panoramas offer an exciting opportunity for improving image acquisition, enabling high-resolution images with minimized perspective distortion. This ensures that climbing walls appear relatively flat in images, enhancing the accuracy of feature and route detection. Additionally, drones allow for efficient coverage of large areas, significantly improving data quality and consistency compared to ground-based methods.

User testing with SAC members will play a vital role in refining and validating the system. Their feedback will help identify bugs, improve usability, and ensure scalability, ensuring the tool meets the real-world needs of climbers. These steps pave the way for significant advancements in climbing route documentation and accessibility.

While these possibilities remain conceptual, they highlight the transformative potential of the project. By continuing to innovate and adapt, this work could contribute to a more comprehensive approach to climbing route management, strengthening SAC's leadership in the climbing community.

2 Table of Contents

I Technical Report

I 1 Introduction

This document presents a comprehensive overview of the project, detailing the various stages from the initial concept to the final results. The aim is to provide a clear understanding of the methodologies employed, the challenges encountered, and the solutions developed. The following chapters will delve into the state of the art, evaluation criteria, implementation concepts, and the results achieved, resulting in a discussion on the evaluation and future outlook of the project.

The documentation is organized to address different needs and audiences. The report serves as a detailed yet accessible account of the project, offering insights suitable for general readers. For developers, the software documentation provides a technical perspective, focusing on implementation and usage details. The project documentation emphasizes the management aspects of the work, covering timelines, roles, and methodologies. Supporting materials are included in Appendix A, which provides additional information, while Appendix B offers a comprehensive reference of tables, figures, and code snippets. This structure ensures that the document remains coherent and accessible to a diverse range of readers, from stakeholders to technical contributors.

I 1.1 Problem Statement

Switzerland's 47,000 climbing routes are scattered across various online platforms and guidebooks, creating challenges in route documentation and accessibility. The manual process of digitizing climbing routes is time-consuming and is leading to outdated or incomplete information. The lack of a centralized, complete and digital system for route management hinders the climbing community's ability to maintain and share accurate route data.

I 1.2 The Vision

The vision for this project is to lay the groundwork for a digitization system that can transform how climbing routes are documented, shared, and maintained. By combining advanced machine learning techniques with user-centered design, the system aspires to create an efficient, semi-automated solution for detecting and organizing climbing infrastructure.

This project aims to simplify the traditionally manual process of identifying and mapping bolts and anchors within climbing routes, thus enhancing accessibility for climbers and route managers. By accurately detecting bolts and anchors from climbing area images, the system provides a robust foundation for route digitization that could one day extend to all Swiss climbing regions and beyond. Furthermore, a simple, intuitive user interface allows for human verification and adjustment, ensuring accuracy and enhancing the flexibility of route management.

In the long term, this proof of concept could serve as a building block for a fully automated, scalable climbing route management solution, integrated with existing platforms such as the Swiss Alpine Club (SAC) tour portal. By capturing and digitizing route data efficiently, this system has the potential to foster safer, well-documented climbing areas, and elevate the climbing community's experience in Switzerland and worldwide.

I 1.3 Industry Partner

This project is being conducted in collaboration with the Swiss Alpine Club, one of Switzerland's largest and most traditional sports organizations. With around 174′726 members [1] and 110 sections [2] spread across the country, the SAC plays a central role in promoting mountain sports, preserving alpine culture, and supporting sustainable tourism in Switzerland. The club manages an extensive network of mountain huts, climbing routes, and hiking trails, making it an indispensable resource for outdoor enthusiasts.

A key component of the SAC's digital offerings is the touring portal, a comprehensive platform that provides climbers, hikers, and mountaineers with access to detailed information about routes, conditions, and infrastructure. However, many climbing routes still lack digital documentation, creating gaps in the SAC's touring portal and limiting the accessibility of these resources to its members.

This project serves as a proof of concept to explore how modern technologies can streamline the digitization of climbing routes and infrastructure. By integrating object recognition and route-mapping tools, the SAC can gain valuable insights into how such technologies might be leveraged to enhance the touring portal. These learnings can inform the SAC's strategy for improving the accuracy and comprehensiveness of their climbing route data, simplifying the process of maintaining and updating the platform. Ultimately, this project highlights innovative approaches to making climbing resources more accessible, while providing a foundation for future developments that support the SAC's mission of preserving and promoting Swiss climbing culture in a digital era.

Figure 4: Icon of SAC: Schweizer Alpen-Club

I 1.4 Objectives

The primary objective of this project is to develop a proof-of-concept system that accelerates the process of digitizing climbing routes by automating the detection of climbing infrastructure from images. This involves several key tasks, including gathering and annotating a dataset of climbing infrastructure images, followed by preprocessing the data to prepare it for training a machine learning model. The model will be trained to detect climbing infrastructure, and an algorithm will be implemented to create the climbing routes.

In addition to these technical objectives, the project aims to evaluate the model's performance using a test dataset and to design a user interface that supports route visualization and management. Finally, the machine learning model will be integrated seamlessly with the user interface to provide a cohesive and functional system. These objectives are specified as functional and non-functional requirements, which are further detailed in the Chapter II 2.

I 1.5 Conditions, Definitions and Boundaries

This chapter outlines the conditions, definitions, and boundaries of the project. It provides a clear understanding of the project's scope, limitations, and requirements to ensure a successful outcome.

I 1.5.1 Conditions

According to the module description [3], the thesis should demonstrate the ability to solve problems using engineering methods and has a conceptual, theoretical, and practical component. The successful completion of the thesis results in 12 ECTS points, which corresponds to a workload of approx. 360 hours per person.

The start date is 17 September 2024, and the submission deadline is 10 January 2024 at 17:00.

I 1.5.2 Definitions

Since the idea of the project revolves around the bolts and anchors that are found on walls, the routes will be limited to those that are equipped with bolts. **The routes will approximate the path of the climber based on the bolts**, as actual routes are mostly drawn beside the bolts and pass through sections where the climber can hold on to. Finding routes that show the path of the climber based on holds and magnesium marks is described as future work in Chapter I 5.3.

I 1.5.3 Boundaries

The project will not cover traditional climbing routes that require the use of removable protection such as nuts and cams.

The proof-of-concept is scoped to the climbing sector "Chämiloch" in Schwyz (see Figure 5) due to its accessibility, suitability for dataset preparation, and well-documented routes. This sector is well equipped with modern bolts and anchors, features a large number of routes, and offers mostly vertical climbing paths that do not cross over each other. Its short and easy approach by train, uncrowded environment, and clear layout with flat walls make it an ideal choice.

The route-finding algorithm is designed specifically for walls with a consistent front-facing orientation (flat walls). Walls with significant overhangs, steep angles, or other irregularities are outside the project's scope.

The sector consists of two distinct rock walls (see Figure 5), enabling the allocation of one wall for model training and the other for validation and testing. The wall marked with a red line is excluded due to its limited space near the railroad, which restricts effective photographing. This clear separation between training and validation walls ensures robust evaluation and reliable results.

The lighting conditions for capturing images are also crucial. Images should be taken in natural daylight without harsh shadows or overexposure to ensure the model can accurately interpret the features. Avoid taking pictures during early morning or late afternoon when shadows are long, and ensure the wall is evenly lit.

Additionally, specific boundaries have been set for the prediction process regarding how the images must be captured and prepared. These guidelines are detailed in the user manual's image-taking tutorial (see Chapter II 7.3), ensuring consistency and quality in the dataset used for model training and validation.

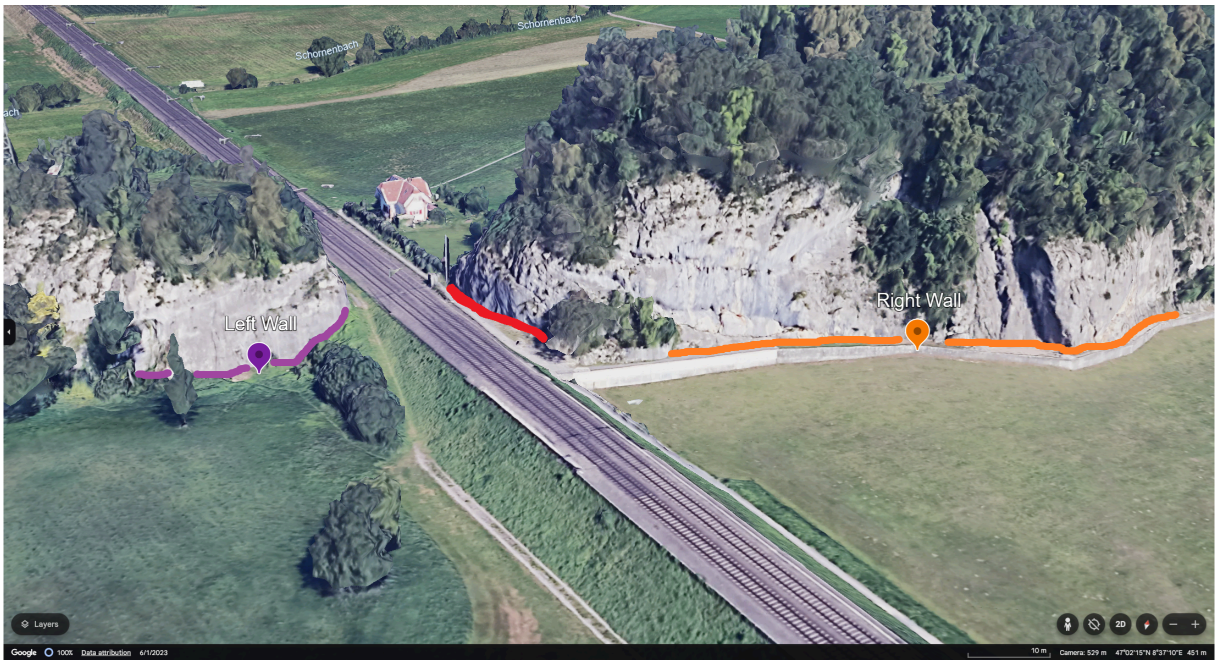


Figure 5: Chämiloch Sector in Schwyz, Switzerland, Choice for Project Scope. (Source: Google Earth)

Further technical details on the project's scope can found in Chapter II 2.1.

I 1.6 Approach and Work Structure

The BoltFinder project adopts an agile approach with a focus on flexibility and iterative development. Scrum methodology structures the work into two-week sprints, ensuring regular feedback and adaptation. This process allows the team to address challenges effectively while aligning progress with the dynamic requirements of the project.

I 1.6.1 Project Framework and Methodology

The project follows an agile development process centered around Scrum, facilitating consistent planning, review sessions, and weekly feedback integration from supervisors. A modular design philosophy is employed, enabling the independent development and seamless integration of system components. Within the team, roles are clearly defined. Nick Wallner concentrates on dataset creation and YOLO model training, also serving as the Scrum Master, while Fabio Elvedi focuses on Faster R-CNN model training and the implementation of the inference API. Both team members work collaboratively on backend and frontend development to deliver a cohesive system.

I 1.6.2 Key Challenges

The project faces several challenges that shape its development. Detecting small objects like climbing bolts and anchors proved difficult due to their size relative to the expansive rock wall images. Additionally, the absence of standardized rules for route design makes generalizing the pathfinding algorithm challenging. Efficiently managing large images for annotation and route editing present further technical demands, requiring innovative solutions.

I 1.6.3 Requirement Definition

Functional and non-functional requirements for the system are derived from the task definition provided by Lars Herrmann and supplemented by discussions with SAC representatives. These requirements guide the design and implementation of the solution, ensuring it met both technical and user needs.

I 1.6.4 Analysis of Existing Solutions

An analysis of existing tools reveals a lack of solutions designed specifically for outdoor climbing route detection. While some methodologies from indoor bouldering projects are explored, the team ultimately develops a custom approaches tailored to the unique challenges of outdoor climbing.

I 1.6.5 Development and Implementation

The implementation phase involves multiple components working in tandem. YOLOv11 and Faster R-CNN are chosen as the object detection architectures, training is done on custom datasets which can be prepared using Jupyter notebooks. The initial rule-based route-finding algorithm is replaced with a Voronoi diagram-based approach, which divides the rock wall into cells and applies neighbor-based rules to determine routes. Dataset preparation involves capturing high-quality images with various devices, supported by a scanning tutorial to ensure data consistency. Technologies such as Python FastAPI for inference, Express.js for the backend, React for the frontend, Typst for documentation, and the Cantaloupe server for efficient image handling are integrated into the system to deliver a comprehensive solution.

I 1.6.6 Testing and Validation

Testing and validation is conducted on one of the rock walls in the “Chämiloch” sector, focusing on evaluating the model’s performance using a custom validation script. The frontend interface supported user corrections, enabling the addition, removal, movement, and relabeling of detected objects to improve accuracy and usability.

I 1.6.7 Collaboration and Communication

The team maintains effective communication through Microsoft Teams and email, with weekly meetings providing continuous guidance from supervisors. Documentation, code repositories, and tracking tools plays a pivotal role in ensuring alignment with project goals and milestones.

I 1.6.8 Deliverables and Scalability

The project delivered trained object detection models, datasets, a functional inference API, backend and frontend systems, and detailed documentation. The modular design of the system allows for easy upgrades or replacements of components, such as object detection models or pathfinding algorithms, making it scalable and adaptable for future enhancements.

I 1.6.9 Used Technologies

This diagram in Figure 6 shows the technologies used in the project. Below the diagram the purpose of each technology is described and links to the respective sections are provided.

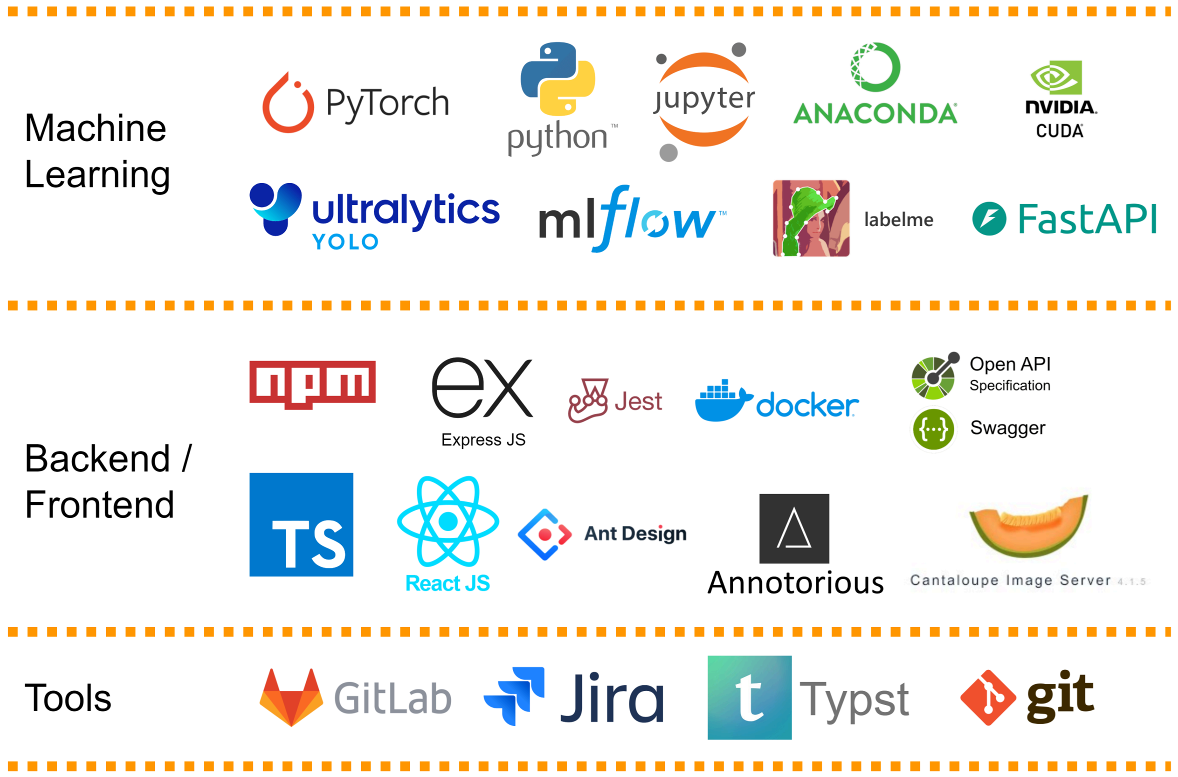


Figure 6: Used Technologies

Frontend: Chapter II 4.4, Chapter II 5.5

- **React JS**: A JavaScript library for building user interfaces, used to create an interactive and responsive UI.
- **Ant Design**: A design system with React components for quickly building consistent and user friendly interfaces.
- **TypeScript**: A strongly typed superset of JavaScript, ensuring type safety in the frontend codebase.
- **Annotorious with OpenSeadragon**: Tools for annotating images within an interactive, zoomable image viewer. (Chapter II 5.5.7)

Backend: Chapter II 4.3, Chapter II 5.4
- **Express JS**: A Node.js framework used to build the server-side logic and API endpoints. <
- **Cantaloupe IIIF Server**: A server for delivering high-resolution images via the IIIF protocol, used for efficient image handling. (Chapter II 4.1.3.2)
- **TypeScript**: Ensures type safety in backend implementation.
- **Jest**: A testing framework for unit and integration tests in the backend. (Chapter II 5.9)

Inference: Chapter II 4.2, Chapter II 5.3
- **PyTorch**: A deep learning framework used for training and deploying machine learning models.
- **Python**: The primary programming language for the inference pipeline and model-related tasks.
- **MLFlow**: A tool for tracking experiments and managing machine learning workflows. (Chapter II 5.2.3)
- **FastAPI**: A fast web framework for serving the machine learning inference API.

Tools:
- **GitLab**: A platform for source control, CI/CD pipelines, and project management.
- **NPM**: A package manager for managing JavaScript dependencies.
- **Jira**: A tool for project tracking and issue management.
- **Docker**: For containerization of applications and ensuring consistent development environments. (Chapter II 5.6)
- **Typst**: A language for documentation as code like LaTeX system used for project documentation.
- **ChatGPT & GitHub Copilot**: AI-powered tools for accelerating code writing and documentation.
- **Mingrammer/Diagrams**: A Python library for creating system architecture diagrams.
- **PlantUML**: A tool for generating UML diagrams.
- **LabelMe Annotation Tool**: Used for labeling images in the dataset. (Chapter II 3.4.1)
- **OpenAPI (Swagger)**: For documenting and testing API endpoints.
- **Anaconda**: A package manager for Python used to manage dependencies and create virtual environments. Used for running python components locally.
- **Jupyter Notebook**: An interactive environment for data analysis and visualization. Used for running trainings and evaluations.

I 2 State of the Art

The state of the art section provides an overview of the existing solutions and norms in climbing route digitization. It outlines the current challenges faced by climbers and the SAC in maintaining climbing route data.

I 2.1 Existing Solutions and Norms

The current state of digitizing climbing routes heavily relies on manual efforts. Climbing guides are predominantly created by hand, with topos manually drawn and data extracted from guidebooks. While this traditional approach has been widely used, it results in fragmented and inconsistent information scattered across various sources, such as online platforms and printed guidebooks.

Prominent platforms like TheCrag, Kletterfux, Vertical Life, and 27Crags provide access to climbing route data. However, these platforms either manage their information manually or rely on community-based contributions. Each platform employs its own unique data format and structure, with variations in topo drawing styles, route grading systems, and metadata for climbing sectors. This lack of standardization adds complexity to the digitization process and limits the interoperability of data between platforms.

Digitizing climbing routes presents several challenges, especially in remote climbing areas, which are often difficult to reach and maintain. Rock climbing sectors vary widely in form, structure, and scale, requiring each topo to be uniquely adapted to the specific features of a climbing wall or area. These challenges underscore the need for innovative approaches to streamline and standardize route digitization.

The SAC has identified key requirements for advancing climbing route documentation. Their primary goal is to evaluate whether a machine learning approach can effectively detect climbing infrastructure in climbing wall images. Additionally, they aim to assess whether the detected infrastructure can be used to identify climbing routes and determine if this approach can be scaled to cover a large number of climbing areas.

From the perspective of climbers, accessibility and usability are crucial. They require an easy way to access route information, including topos and climbing conditions. Topos should be designed to help climbers quickly identify their preferred routes and navigate climbing sectors with minimal effort. Meeting these requirements would significantly enhance the overall climbing experience while addressing the limitations of the current manual approach to route digitization.

I 2.2 Analysis of Existing Projects, Tools and Techniques

This section explores current tools, technologies, and methods relevant to the development of an object recognition model for detecting climbing bolts on mountain routes. The analysis focuses on identifying the most suitable tools and techniques based on prior work in computer vision and machine learning (ML) applications for climbing. Two main areas are examined: (1) object recognition, particularly for climbing holds or bolts, and (2) pathfinding algorithms for detecting routes in outdoor settings.

I 2.2.1 Existing Projects and Approaches

1. **The Rock Climbing Coach project** [4] focuses on analyzing video footage of indoor climbing to produce post-climb reports. The project utilizes computer vision techniques such as Canny Edge Detection and blob detection, alongside machine learning models like YOLO for object detection. Due to challenges with noise and chalk on walls, the project transitioned to a machine learning approach, achieving high accuracy in detecting climbing holds using annotated datasets.

   Relevance: The hold detection method, particularly the machine learning-based segmentation, could be adapted to detect outdoor climbing bolts. The Project augmented the dataset by horizontal, vertical flips and hue changes, enabling to triple the original dataset size.

2. **ClimbSense Project** [5] introduces a system to automatically recognize climbed routes using wrist-worn Inertia Measurement Units (IMUs). The system extracts features of climbers' ascent data and uses them to train a recognition system. A high recognition rate of over 90% is achieved using cross-validation techniques.

   Relevance: While this project focuses on indoor routes and movement patterns, its wearable IMU-based tracking is outside the scope of the current project. However, it could be considered for future work to refine the model by incorporating additional data on climbing motion patterns, complementing image-based bolt detection for enhanced accuracy.

3. **Computer Vision Based Indoor Rock Climbing Analysis** [6] This tool uses Computer Vision and Machine Learning to analyze video footage of indoor rock climbing. Unlike sensor-based devices, it provides post-climb reports by detecting climbing holds with YOLO and tracking climbers' movements using Keypoint Detection APIs. It measures metrics like route completion and moves taken, achieving an average RMSPE of 0.266 with potential for improvement.

   Relevance: The study is relevant as it explores methods for detecting climbing holds and tracking movements indoors using Computer Vision. These techniques could be adapted for our outdoor use case, where natural rock formations and bolts require different detection approaches.

3. **Remote Sensing Object Detection** [7] surveys deep learning-based object detection techniques in remote sensing images. It addresses challenges such as complex backgrounds, small object sizes, and the wide scale variance common in remote sensing images, which are similar to the issues encountered in detecting small climbing bolts on natural rock surfaces. The survey highlights strategies such as multi-scale feature fusion and attention mechanisms to enhance detection accuracy.

   Relevance: These methods could be applied to detect small climbing bolts and reduce background noise in outdoor settings. Multi-scale feature fusion and attention mechanisms may help detect small bolts against complex rock backgrounds, improving accuracy and reducing false positives.

4. **Object detection using deep learning on metal chips in manufacturing** [8] evaluates automated detection and classification of metal chips using CNN models like AlexNet, VGG16, and a custom CNN designed for the specific task. The approach also tests various lighting and angles to optimize image quality for classification.

   Relevance: The use of image preprocessing techniques, such as foreground and edge filters, could be highly beneficial for our project in climbing bolt recognition. These filters would enhance the model's ability to distinguish bolts from complex backgrounds, such as natural rock features, making detection more efficient. While the current focus is on bolt detection, integrating these filtering techniques could improve accuracy and streamline the process, especially in challenging outdoor environments.

4. **Metal Surface Defect Detection Using Modified YOLO** [9] applies a modified YOLO model to detect small defects on metal surfaces, focusing on improving the accuracy of detecting small, hard-to-detect features. Similar articles: [10], [11].

   Relevance: The improvements made in YOLOv3 for detecting small objects in this project could be adapted not only to detect climbing bolts but also to assess their condition, identifying potential wear or damage. However, while this could enhance the system's capabilities, integrating bolt condition analysis is beyond the scope of the current project and could be explored in future work.

5. **Deep learning of rock images for intelligent lithology identification** [12] utilizes deep learning with Faster R-CNN architecture to predict lithology and position information from rock images, improving accuracy through data augmentation and pre-training.

   Relevance: This approach could be valuable for adding metadata about the rock type in climbing sectors. While our current project focuses on detecting climbing bolts, incorporating lithology identification could enrich the data structure for climbers, offering detailed insights into the rock quality. However, this falls beyond the current scope and could be explored in future developments.

I 2.2.2 Datasets and Pre-trained Models

Currently, no preexisting datasets specifically designed for detecting climbing bolts on natural rock surfaces are identified. While some datasets for bouldering or indoor climbing holds exist, they primarily focus on detecting holds in controlled environments, such as climbing gyms, rather than bolts on outdoor climbing routes.

In our research, we found only datasets related to metal detection, which are not directly applicable to our project. For example:

- How to Detect Metal Defects with Computer Vision: This dataset focuses on detecting metal defects in industrial settings, such as identifying scratches or cracks on metal surfaces. [13]
- Detecting Metal in General: This dataset primarily features objects like metal cans and other general metal items, which are not representative of climbing bolts or their unique features in natural outdoor environments. [14]

Given the lack of a suitable dataset for climbing bolt detection, it will be necessary to create a custom dataset. This will involve collecting images of climbing routes on natural rock, annotating climbing bolts, and incorporating data augmentation techniques to address the challenges of detecting small objects in complex outdoor environments.

I 2.2.3 Tools and Techniques

The development of an object recognition model for detecting climbing bolts in outdoor settings draws heavily on advanced tools and methodologies from computer vision and machine learning. These techniques address challenges such as small object detection, complex backgrounds, and varying scales inherent in climbing wall images. The following tools and approaches form the foundation for this project.

The YOLO framework, particularly YOLOv11 and its modifications, has proven highly effective for real-time object detection. Its ability to balance speed and accuracy in complex scenes makes it an excellent choice for detecting small objects like climbing bolts. Modifications to YOLO, such as improved anchor box size selection and multi-scale feature extraction, have been shown to significantly enhance detection accuracy, as demonstrated in studies on metal surface defect detection. These adaptations can be tailored to the unique requirements of climbing bolt recognition, where bolts are small and embedded in intricate rock backgrounds. [15]

Faster R-CNN, another leading architecture in object detection, offers a robust alternative for climbing bolt detection. Known for its two-stage detection process, Faster R-CNN excels in achieving high accuracy, particularly for small objects. By leveraging a region proposal network (RPN) and feature extraction, Faster R-CNN provides detailed bounding box predictions that are especially valuable for detecting bolts in complex outdoor environments. When paired with data augmentation and pre-trained models, Faster R-CNN can address challenges like scale variation and object occlusion, complementing the YOLO-based approach. [16]

Multi-scale feature fusion techniques, commonly employed in remote sensing, enhance object detection by enabling models to recognize objects of varying sizes in high-resolution images. Incorporating multi-scale feature fusion allows the system to detect small bolts across diverse scales, improving both accuracy and robustness in outdoor climbing settings. [17]

Data augmentation is critical for overcoming the lack of pre-existing datasets tailored to climbing bolts. Techniques such as flipping, rotating, and altering image properties (e.g., hue or contrast) expand the dataset by simulating various real-world conditions. These methods improve the model's generalization ability, enabling it to perform reliably under different lighting, angles, and environmental conditions.

Depth recognition can further refine detection accuracy by providing spatial context. Tools like PyTorch MiDaS estimate depth from a single image, helping the model distinguish bolts from similar looking features in cracks or irrelevant areas. For enhanced precision, 3D reconstruction tools such as OpenSfM can create a three-dimensional representation of climbing walls, offering additional verification for bolt positioning and identification. [18]

Pathfinding algorithms are essential for climbing route identification. Voronoi diagrams, commonly used in computational geometry, can divide a climbing wall into cells based on bolt positions, providing a structured way to map potential routes. By applying neighbor-based rules to these cells, routes can be determined in a logical and efficient manner. Rule-based algorithms further refine this process by incorporating predefined criteria, such as bolt spacing or angle restrictions, to ensure that the generated routes align with climbing norms and safety requirements. These techniques offer climbers a clear and accurate visual representation of routes, supporting better planning and enhanced safety. [19]

I 2.2.4 Conclusion

The reviewed projects and techniques provide essential insights and tools for developing an object recognition model tailored to detecting outdoor climbing bolts. Modifications to the YOLO framework and strategies like multi-scale feature fusion are especially promising for improving detection accuracy in complex environments. Additionally, the integration of wearable IMU sensors or lithology detection could further enhance the benefits of using a visual recognition model for digitizing climbing routes.

I 2.3 Deficiencies and Improvement Potential

The current practice of managing and documenting outdoor climbing routes poses significant challenges. These include a lack of standardized approaches, inconsistent data quality across regions, and a reliance on manual methods that are time-consuming and prone to error. In addition, existing object detection methods are often optimized for controlled environments, limiting their effectiveness in detecting small objects such as climbing bolts on natural and irregular rock surfaces.

The complexity of outdoor climbing data, such as the variability of terrain and the need for scalable systems to process diverse information, highlights the limitations of current methods. Furthermore, the lack of comprehensive and specialized datasets for outdoor climbing applications underscores the need for tailored solutions.

This project seeks to address these gaps by utilizing machine learning for bolt detection, path-finding algorithms for accurate route mapping, and providing tools for scalable and efficient route documentation. By addressing these specific shortcomings, the project provides a transformative approach to climbing route management that meets the needs of climbers and supports organizations like the SAC.

I 3 Evaluation Criteria

The evaluation criteria define the metrics and benchmarks used to assess the performance and effectiveness of the project. The project evaluation relies on defined metrics that address its machine learning model performance, the effectiveness of the route-finding algorithm, and the usability of the developed interface. These criteria are essential for measuring the success of the project and determining the areas that require improvement. They provide a structured approach to assess how well the project objectives are met.

I 3.1 Machine Learning Model Performance

The performance of the ML model is measured by its ability to predict the presence of climbing bolts and anchors in high-resolution, panorama-like images. Key metrics include:

- **F1 Score, Precision, and Recall**: These determine the accuracy and completeness of detection, with recall prioritized to ensure bolts and anchors are detected comprehensively, even at the risk of false positives.

$$\mathrm{F1} = 2 \cdot \frac{\mathrm{Precision} \cdot \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}} \qquad \mathrm{Recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \qquad \mathrm{Precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

TP = True Positives, FP = False Positives, FN = False Negatives

- **Mean Average Precision (mAP)**: Scores such as mAP-50 and mAP-50-95 evaluate precision across varying intersection over union thresholds, with higher values indicating better model performance in detecting objects of different sizes and complexities.

$$\mathrm{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

- **Inference Time**: While speed is not a critical focus in this project, reasonable inference times improve the model's practical usability, particularly during field applications.

I 3.2 Route Algorithm Evaluation

The performance of the route algorithm is evaluated by its accuracy in generating realistic and coherent climbing routes. Accuracy reflects how well the algorithm translates machine learning predictions into valid routes and is assessed using the following categories:

- **Correctly Identified Routes**: Routes where all bolts and anchors are connected exactly as expected, with no missing or extra elements.
- **Minor Errors**: Routes with up to one mistake, such as a single missing or extra bolt, which does not significantly affect the overall structure of the route.
- **Major Errors**: Routes with more than one mistake, such as multiple missing or extra bolts or incorrect connections, rendering the route impractical or unrealistic.

I 3.3 User Interface Usability

The prediction and route editor's effectiveness is assessed by its alignment with the functional requirements outlined in the documentation. The interface must support user tasks like adjusting, verifying, or supplementing the detected routes. Key parameters are:

- **Ease of Use**: The interface must be intuitive, reducing the cognitive load on users performing manual corrections or edits.
- **Completeness of Features**: The editor should allow users to add, move, remove, or relabel bolts and anchors and visually map the routes.

I 3.4 Importance in the Context of the Project

These criteria are critical for validating the feasibility of the proof-of-concept. High performance in bolt detection ensures the foundational data is reliable, while robust route algorithms and user-friendly interfaces enable the system to fulfill its goal of assisting climbers and route maintainers. Each criterion aligns with the project's overarching aim of automating the digitization of climbing routes while maintaining accuracy and flexibility for manual interventions.

I 4 Implementation Concept

The implementation concept outlines the conceptual and technical solutions developed to address the project requirements. The BoltFinder system aims to streamline the digitization of climbing routes by automating the detection of climbing infrastructure and generating route data structures. It is divided into three core components: a machine learning model for detecting bolts and anchors, a route generation algorithm, and a user interface for visualization and management.

I 4.1 System Architecture

The system's architecture is visualized in the following C4 diagram in Figure 7, providing an overview of the key components and their interactions. The architecture includes a machine learning model for bolt detection, a backend service for processing predictions and generating routes, and a frontend interface for user interaction. The backend communicates with the inference server to fetch model predictions and processes the data for route visualization.

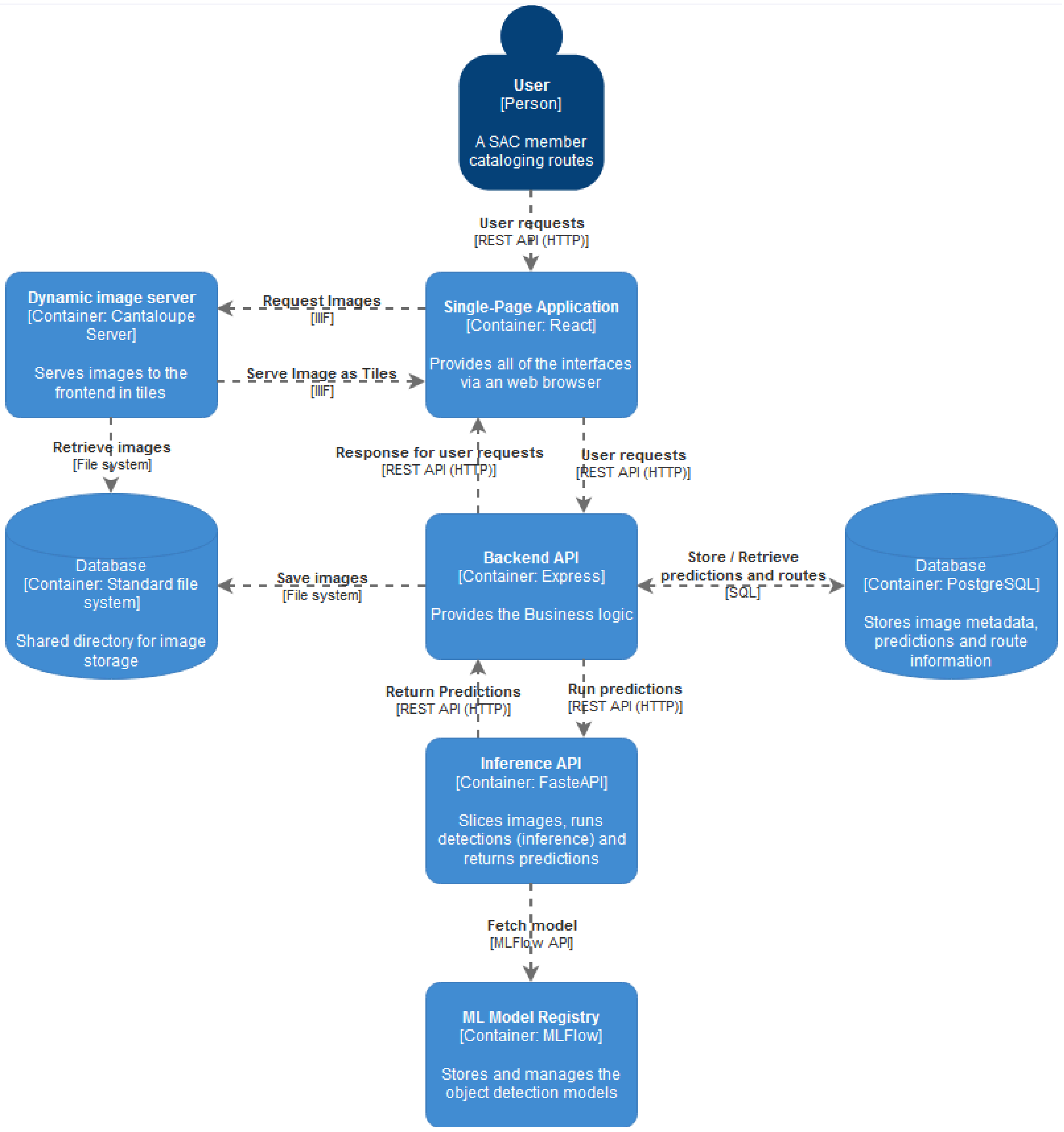


Figure 7: C4 Architecture Overview of the BoltFinder System

The high level data flow diagram illustrated in Figure 8 depicts the flow of data within the BoltFinder system. There are two main processes in the system: the training of the machine learning model and the generation of topos. The training process includes the preparation of the dataset, training the model, and saving the model. The generation of topos includes the prediction of bolts and anchors, the algorithmic detection of routes and the editing functionality for predictions and routes.

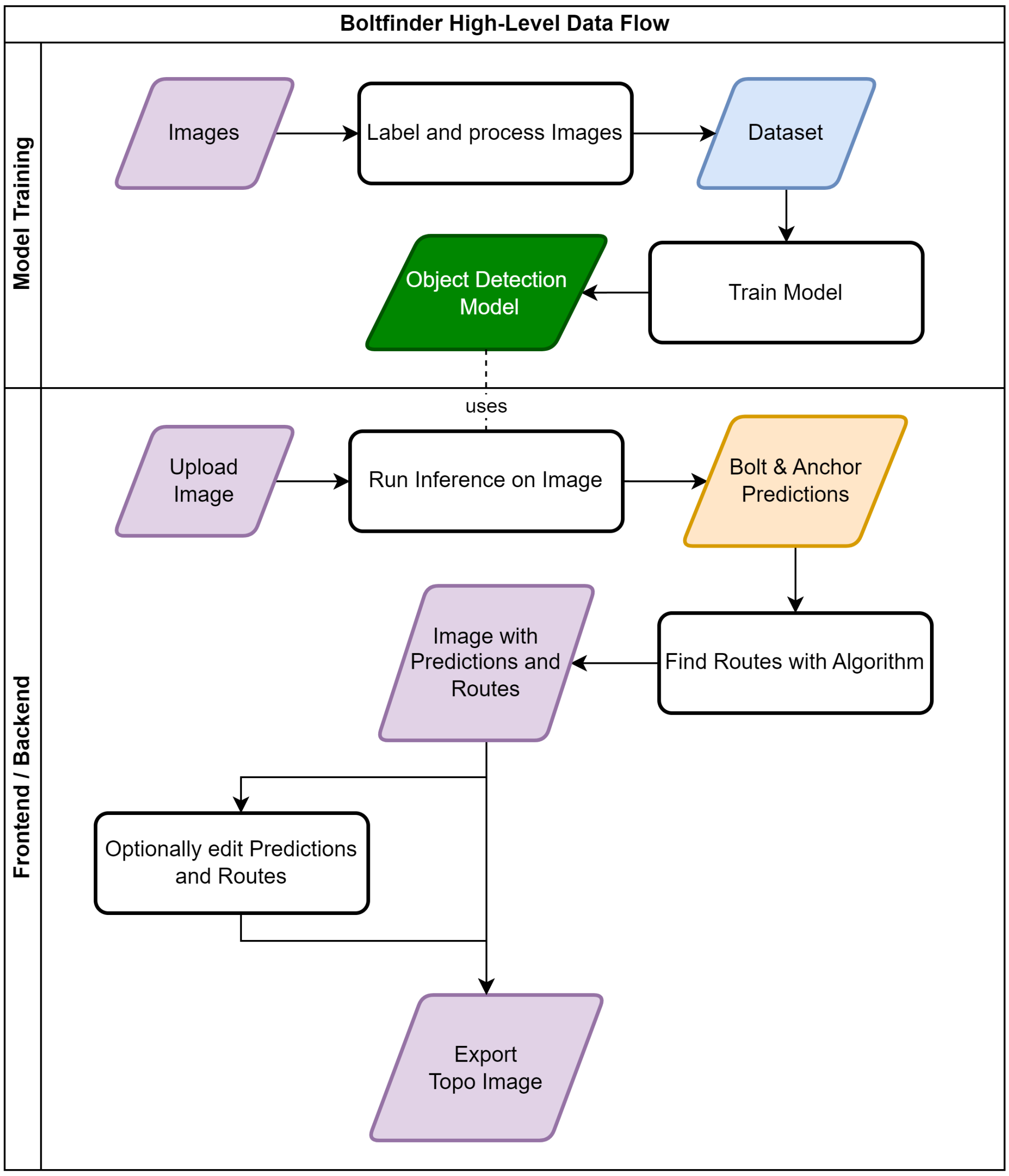


Figure 8: Data Flow within the BoltFinder System

More detailed information about the individual steps and components are found in the software documentation chapters: Chapter II 4 and Chapter II 5

I 4.2 Machine Learning Model

The machine learning model is trained to detect bolts and anchors in high-resolution panorama images. It relies on a dataset of annotated images created for this project and employs advanced object detection techniques to ensure high precision and recall. Model predictions serve as input for the route generation algorithm.

I 4.3 Route Generation Algorithm

Using predictions from the ML model, the route generation algorithm creates a data structure representing climbing routes. It utilizes computational geometry techniques, such as Voronoi diagrams and rule-based connections, to generate realistic and accurate routes. User edits are incorporated through the creation of white and blacklists, which the algorithm considers during recalculations to refine and adapt the generated routes based on manual adjustments.

I 4.4 User Interface

The system's user interface allows users to visualize and manage generated routes. It enables editing predictions, including adding, removing, moving, and changing labels of bolts and anchors. Users can also edit routes by adding or removing them, and modifying which bolts and anchors belong to each route. Images are loaded from a Cantaloupe server over the IIIF protocol as tiles and displayed to the user using the OpenSeadragon library, enabling efficient handling and interactive viewing of high-resolution climbing wall images.

I 4.5 Integration and Scalability

The modular design of the system allows seamless integration between the ML model, backend services, and frontend interface. This structure ensures flexibility for future enhancements, such as incorporating additional climbing features or expanding the system's scope to new climbing areas.

I 5 Results, Evaluation, and Outlook

This chapter covers the results of this work and discusses whether the objectives are met. Furthermore, the remaining bugs and recommendations for extensions of the BoltFinder implementation are shown. Finally, general improvement possibilities of the web application are described.

I 5.1 Results

The BoltFinder project successfully develops a proof-of-concept system that automates the detection of climbing bolts and anchors, streamlining the digitization of climbing routes. This section summarizes the system's performance, including key metrics and visual results from its implementation.

I 5.1.1 Summary of Requirements Fulfillment

The BoltFinder system successfully meets 7 out of 8 functional requirements, including accurate detection of climbing bolts with an F1 score of 91% and a recall above 90% (Figure 121), reliable route generation with only 1% major errors (Figure 128), and full support for manual adjustments, additions, deletions, merging, and visualization of routes. One requirement, related to user-friendliness, is not fulfilled due to the lack of a formal user study.

For non-functional requirements, the system fulfills 3 out of 4, ensuring that users are informed if actions take longer than 2 seconds, demonstrating modularity that allows independent updates, and compatibility with defined data structures. However, user error protection is only partially achieved due to missing confirmation modals for some destructive actions.

Further details on the fulfillment of functional and non-functional requirements are provided in the following Chapter II 6.1.3.

I 5.1.2 Dataset Creation and Preprocessing

A custom dataset of 832 images including 3'995 bolts and 269 anchors is created using high-resolution climbing wall images. The images are annotated with bolt and anchor locations. To improve model generalization, data augmentation techniques such as flipping, rotation, and color adjustments are applied during training.

Further details on the dataset creation and preprocessing are provided in the following Chapter II 5.1.

I 5.1.3 Machine Learning Model Performance

The project evaluates YOLOv11 and Faster R-CNN models, for detecting bolts and anchors. YOLOv11 emerges as the best-performing model. The following predictions in Figure 9 show the model's ability to detect bolts and anchors accurately:

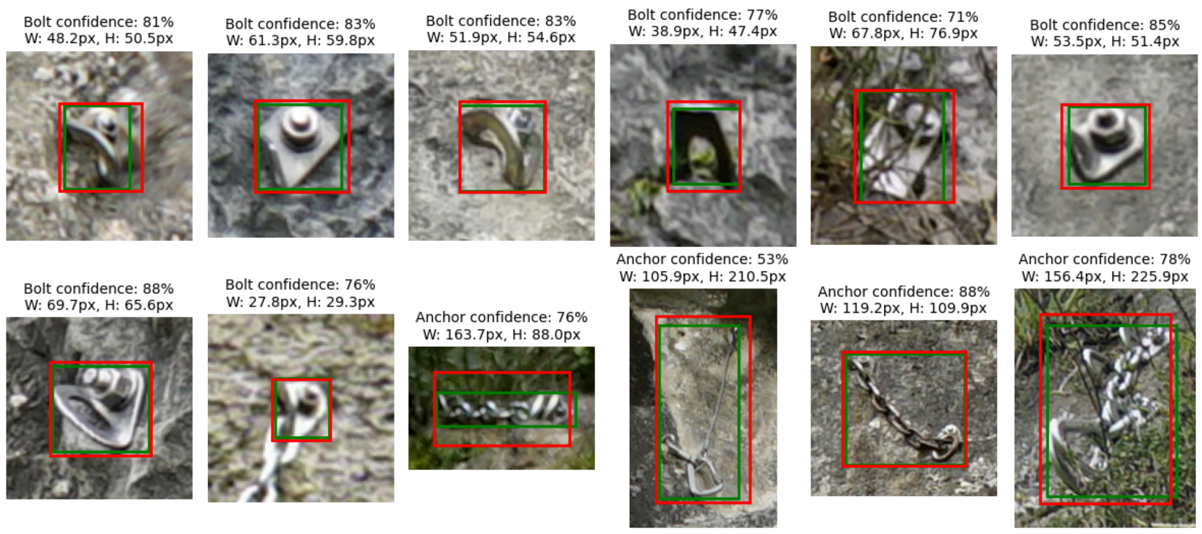


Figure 9: Predictions from the YOLOv11 Model (trained and tested on same data)

The F1 scores the Model trained and tested with all available data are shown in the following Figure 10. The maximum F1 score is 91% for bolts and 87.9% for anchors. These results demonstrate the model's ability to balance precision and recall effectively and achieve high precision at a low confidence threshold.

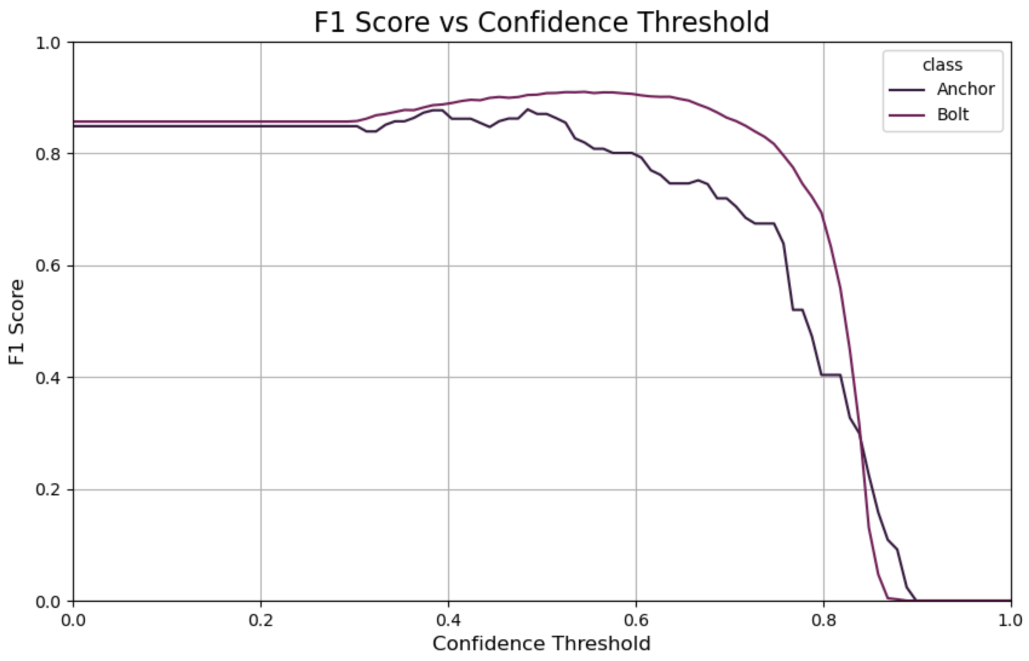


Figure 10: F1 vs Confidence Threshold for YOLO Model (trained and tested on same data)

Further details on the model performance are provided in the following Chapter II 6.1.1.4.

I 5.1.3.1 YOLOv11 vs Faster R-CNN

The comparison of a YOLOv11 and Faster R-CNN reveals that YOLOv11 outperforms Faster R-CNN in precision, reducing false-positive detections significantly.

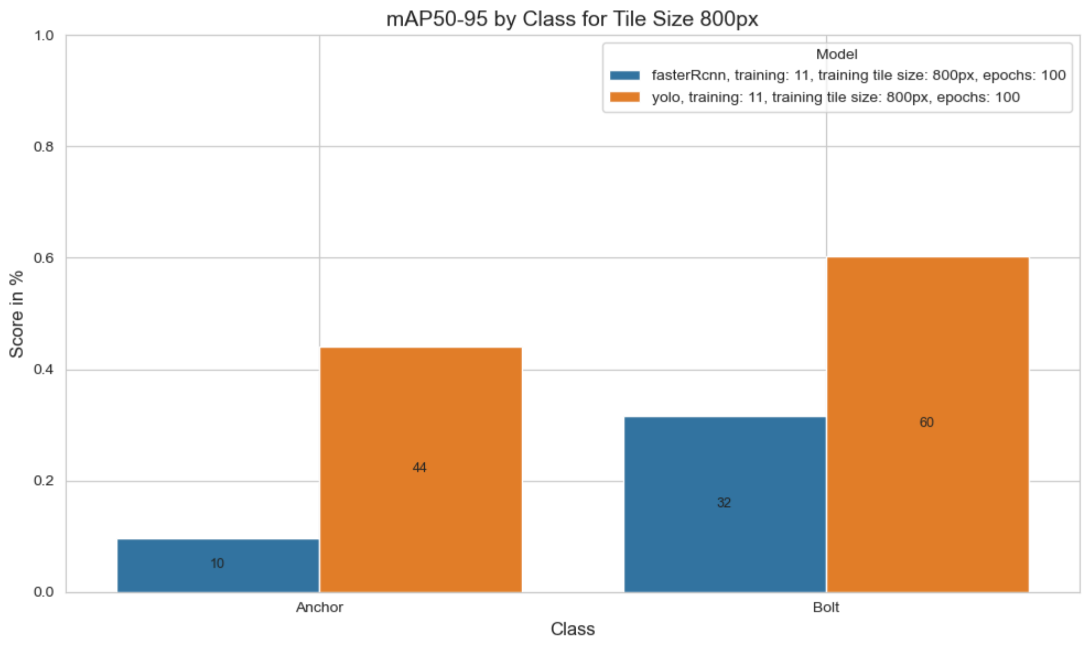


Figure 11: Comparison of mAP-50-95: YOLOv11 vs. Faster R-CNN

Visual comparisons of mAP and F1 scores highlight this distinction, as shown in Figure 11 and Figure 12.

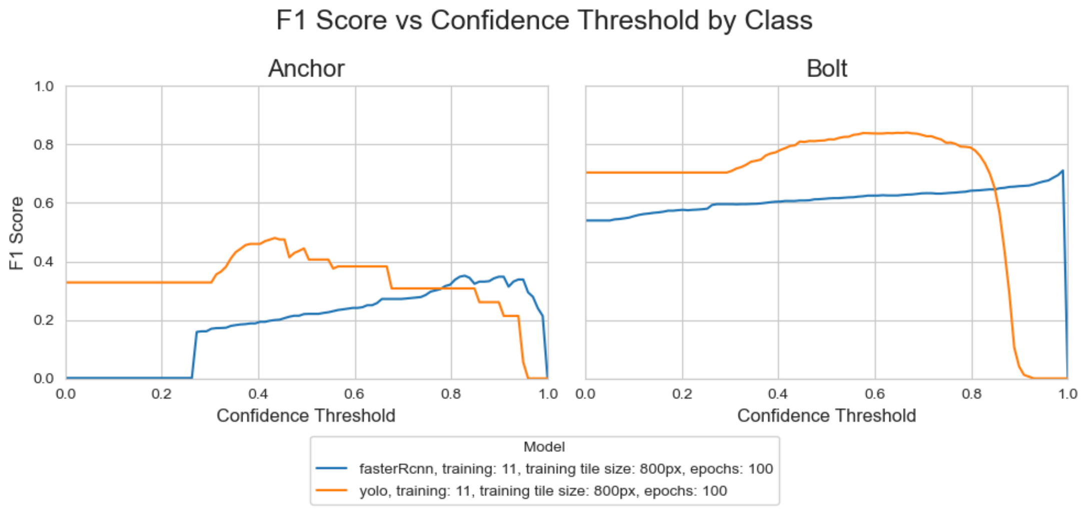


Figure 12: Comparison of F1 Scores: YOLOv11 vs. Faster R-CNN

Further details on the comparison between YOLOv11 and Faster R-CNN are provided in the following Chapter II 6.1.1.1.

I 5.1.4 Route Generation Algorithm

The route generation algorithm uses Voronoi diagrams and rule-based techniques to map bolts into realistic climbing routes. Testing on 102 routes produces the following results:

- **Correctly Identified Routes**: 78% of routes match expected connections.
- **Minor Errors**: 21% of routes contain slight deviations, such as missing or extra bolts.
- **Major Errors**: Only 1% of routes show significant inaccuracies.

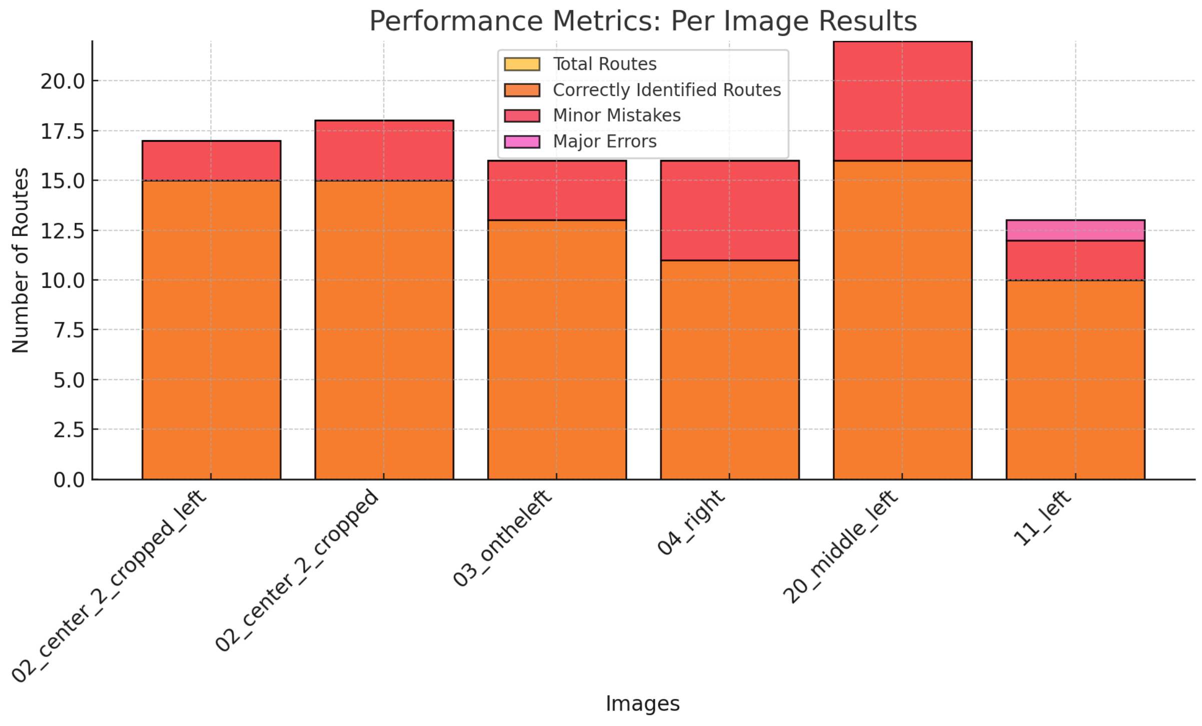


Figure 13: Route Detection Accuracy

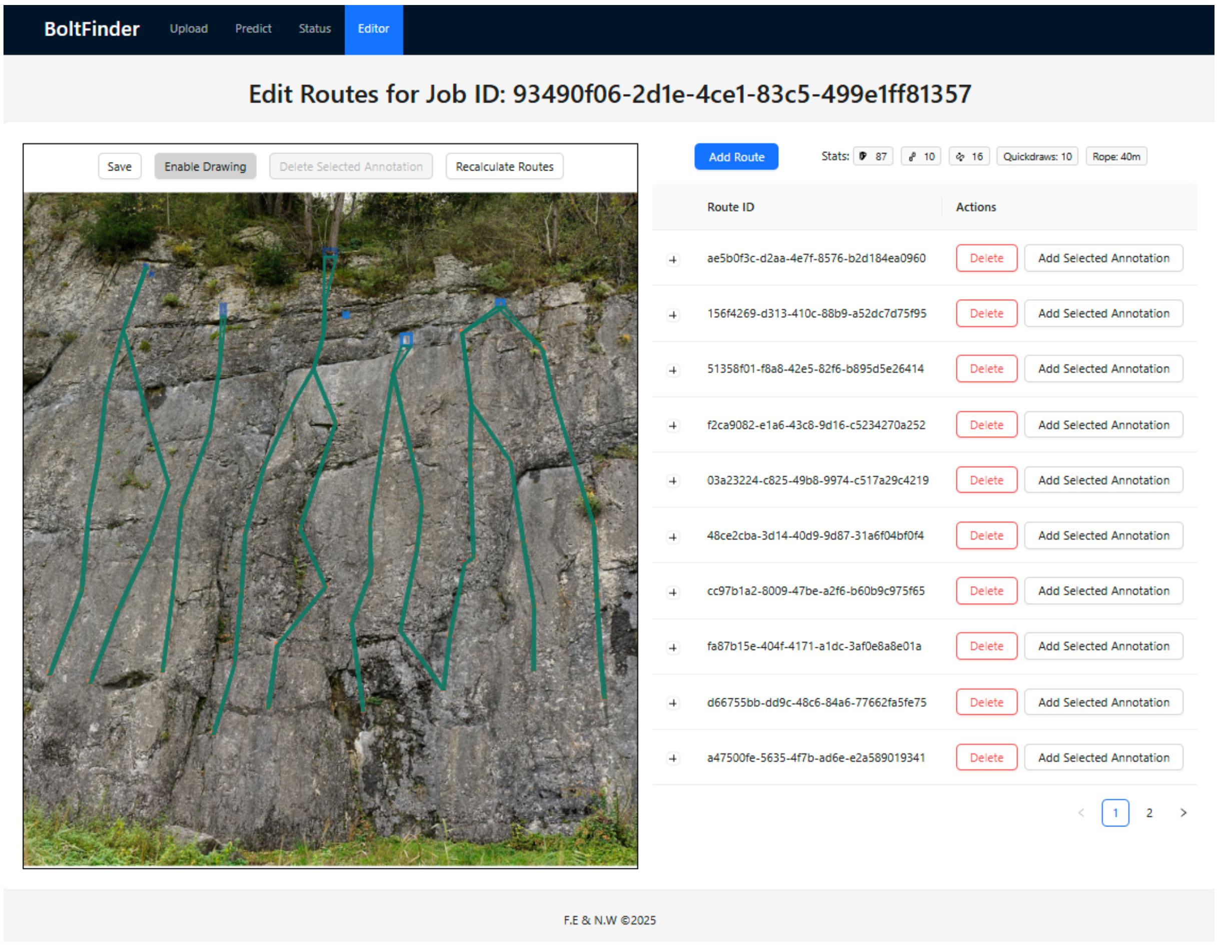
Further details on the route generation algorithm's performance are provided in the following Chapter II 6.1.2.

I 5.1.5 User Interface and Usability

A user-friendly interface is developed for route visualization and management. It supports:

- High-resolution image uploads for prediction generation on images with 25′000px in width and height.
- Running prediction jobs with customizable inference settings.
- Editing and refining predictions by adding, removing, or repositioning bolts and anchors.
- Editing routes by merging, splitting, or deleting routes.
- Downloading the final topology image with bolts, anchors, and routes.

The following Figure 14 shows the BoltFinder editing interface, highlighting the tools available for route adjustments and corrections.



Edit Routes for Job ID: 93490f06-2d1e-4ce1-83c5-499e1ff81357

Stats: 87 10 16 Quickdraws: 10 Rope: 40m

	Route ID	Actions
+	ae5b0f3c-d2aa-4e7f-8576-b2d184ea0960	Delete Add Selected Annotation
+	156f4269-d313-410c-88b9-a52dc7d75f95	Delete Add Selected Annotation
+	51358f01-f8a8-42e5-82f6-b895d5e26414	Delete Add Selected Annotation
+	f2ca9082-e1a6-43c8-9d16-c5234270a252	Delete Add Selected Annotation
+	03a23224-c825-49b8-9974-c517a29c4219	Delete Add Selected Annotation
+	48ce2cba-3d14-40d9-9d87-31a6f04bf0f4	Delete Add Selected Annotation
+	cc97b1a2-8009-47be-a2f6-b60b9c975f65	Delete Add Selected Annotation
+	fa87b15e-404f-4171-a1dc-3af0e8a8e01a	Delete Add Selected Annotation
+	d66755bb-dd9c-48c6-84a6-77662fa5fe75	Delete Add Selected Annotation
+	a47500fe-5635-4f7b-ad6e-e2a589019341	Delete Add Selected Annotation

< 1 2 >

Figure 14: BoltFinder Editing UI

I 5.1.6 Summary of Achievements

The BoltFinder system demonstrates the feasibility of automating the digitization of climbing routes. Its machine learning models deliver high accuracy in bolt and anchor detection, while the route generation algorithm successfully maps realistic routes. The interface enhances usability, paving the way for future scalability and integration with platforms like the SAC touring portal.

By integrating advanced machine learning techniques, computational geometry, and user-centric design, BoltFinder provides a robust solution for climbing route documentation, supporting climbers and route maintainers alike.

The below Figure 15 shows a final topology image generated by the BoltFinder system, illustrating the combined output of bolt and anchor detection and route generation.

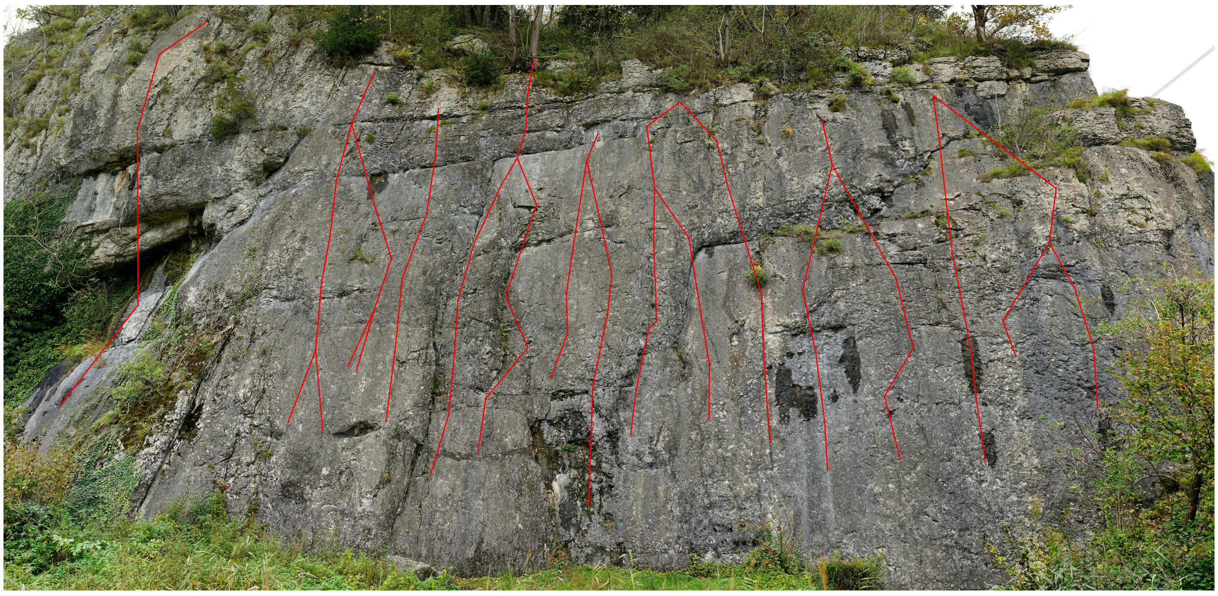


Figure 15: Final Topology Image Generated by BoltFinder

I 5.2 Bugs and Limitations

While the proof-of-concept achieves its primary objectives, several limitations and areas for improvement are identified during testing.

I 5.2.1 Orientation-Related Limitations

Performance for route detection varies significantly based on wall orientation, as shown in Figure 16. The two images depict the same wall: the left image shows the generated climbing routes, while the right image displays the actual wall. The green section represents the most front-facing wall orientation, achieving the highest accuracy. The yellow section corresponds to slightly angled walls, introducing minor inaccuracies. The red section highlights steep walls facing away, where the algorithm struggles the most, leading to routes starting in the middle of the wall or bolts not connecting to nearby bolts.

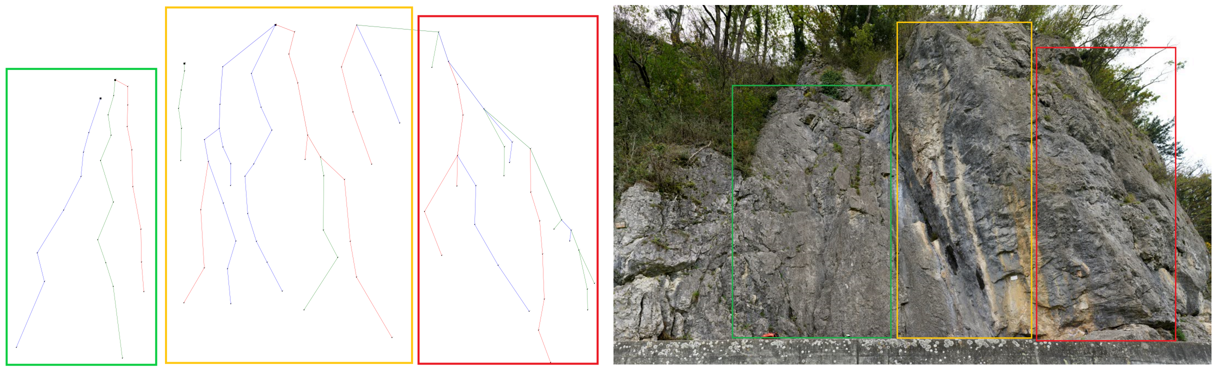


Figure 16: Route Detection Accuracy across Wall Orientations. Left: Generated Climbing Routes. Right: Actual Wall with Sections highlighted in green (front-facing), yellow (angled), and red (steep).

I 5.2.2 Route Detection Challenges

In addition to orientation-related limitations, route detection faces challenges in accurately representing complex paths, especially when routes converge or overlap. These cases often lead to ambiguities in the generated routes, such as misaligned connections or missing segments. Refining the route generation algorithm to better handle such scenarios would improve overall accuracy and reliability, especially for climbing routes on intricate or crowded walls.

I 5.2.3 Bolt and Anchor Detection

Bolt and anchor detection is not always consistent, particularly for anchors, due to the limited number of annotated examples in the dataset. The model often misidentifies bolts, mistaking objects such as branches or trees for climbing infrastructure. Addressing this issue could involve preprocessing images to mask background areas, reducing false positives.

I 5.2.4 Scalability of the FastAPI Server

From a technical perspective, the FastAPI server demonstrated limited scalability during testing. It can handle up to five concurrent prediction requests for panorama images with an 800px tile size and 0.3 overlap. Beyond this threshold, the server may become unresponsive until processing threads are freed. This limitation also affects the prediction UI, where the model dropdown relies on the server's availability. Addressing these issues would improve the system's robustness and scalability for larger-scale deployments.

Despite these challenges, the BoltFinder system provides a solid starting point for future development, with clear avenues for refinement and enhancement.

I 5.3 Outlook

The proof-of-concept has demonstrated the feasibility of detecting bolts, anchors, and routes in climbing sectors, providing a strong foundation for future development. Expanding the system's capabilities presents exciting possibilities for enhancing its functionality and usability.

One promising direction is the use of drones to capture high-quality panoramic images of climbing sectors. This would improve data acquisition, especially in remote or hard-to-reach areas. Future work could also focus on training the model to recognize additional bolt types and to analyze the condition of bolts, such as identifying signs of rust or wear. Expanding the detection capabilities to include climbing holds and predicting climbing grades could make the system even more valuable for climbers.

Incorporating features like estimating route height and steepness would add another layer of detail to the route data, enhancing its utility. Integration with the SAC touring portal could streamline data sharing and access for climbers and route maintainers. Additionally, implementing bulk operations for uploading and processing multiple images at once would make the system suitable for digitizing entire climbing sectors efficiently.

Another area of improvement involves optimizing the prediction process by identifying and excluding background elements such as trees, bushes, or other irrelevant areas. By overlaying an exclusion map on the input image, the system could focus solely on relevant parts of the climbing wall, reducing computational overhead and improving accuracy.

These enhancements highlight the potential for BoltFinder to evolve into a comprehensive tool for climbing route documentation, catering to both recreational climbers and professional organizations.

II Software Documentation

II 1 Vision

The vision is described in Chapter I 1.2

II 2 Requirement Specification

This chapter provides a comprehensive overview of the requirements and scope for the software project. It includes both functional and non-functional requirements, detailing the necessary specifications that the system must fulfill. The requirements are categorized and described to ensure clarity and facilitate the development process.

II 2.1 Scope

This chapter further outlines the scope of the project like described in Chapter I 1.4 & Chapter I 1.5. The system under development focuses on detecting and organizing climbing infrastructure, specifically bolts and anchors, within rock wall images. The project primarily operates on data from the climbing area **Chämiloch**, utilizing images that include single bolts/anchors or entire climbing routes. The scope of this work is limited to developing a proof of concept (PoC) rather than a full production-level solution.

Key focus areas within the scope include:

1. Detection of bolts and anchors: Training a machine learning model to detect and accurately identify bolts and anchors in climbing images.
2. Route Identification: For images containing multiple climbing routes, assigning bolts and anchors to their respective routes.
3. Manual Editing UI: Providing users with a simple user interface to manually adjust and correct bolt/anchor detection, and reassign bolts or anchors to routes if necessary.
4. Creating a Climbing Topo: Mapping the computed climbing routes back onto the original image through a simple line.

> **Important**
>
> A limitation of using bolts and anchor positions is that the identified routes correspond specifically to the locations of bolts and anchors on the rock wall where climbers secure their rope onto. Unlike climbing guidebooks that illustrate the full path a climber should follow, this system focuses exclusively on detecting safety equipment anchor points, not the complete climbing route.
>
> The Chapter I 5.3 describes potential future enhancements and extensions to address this limitation.

The proof of concept will provide a working prototype that validates these functionalities using real data from the climbing garden “Chämiloch” see (see Figure 5), but it is not intended as a complete or fully scalable solution.

II 2.2 Actors

Actor	Description
Climber	The end user of the system who uses the app or software to identify bolts and anchors and view climbing routes on a rock wall. Their main interest is safety, clarity, and the accuracy of the detected climbing infrastructure.

Table 1: Actors

II 2.3 Use Case Diagram

The following diagram Figure 17 illustrates the primary use cases and interactions between the actors and the system. It provides a high-level overview of the system's functionalities and the relationships between the actors and the system components.

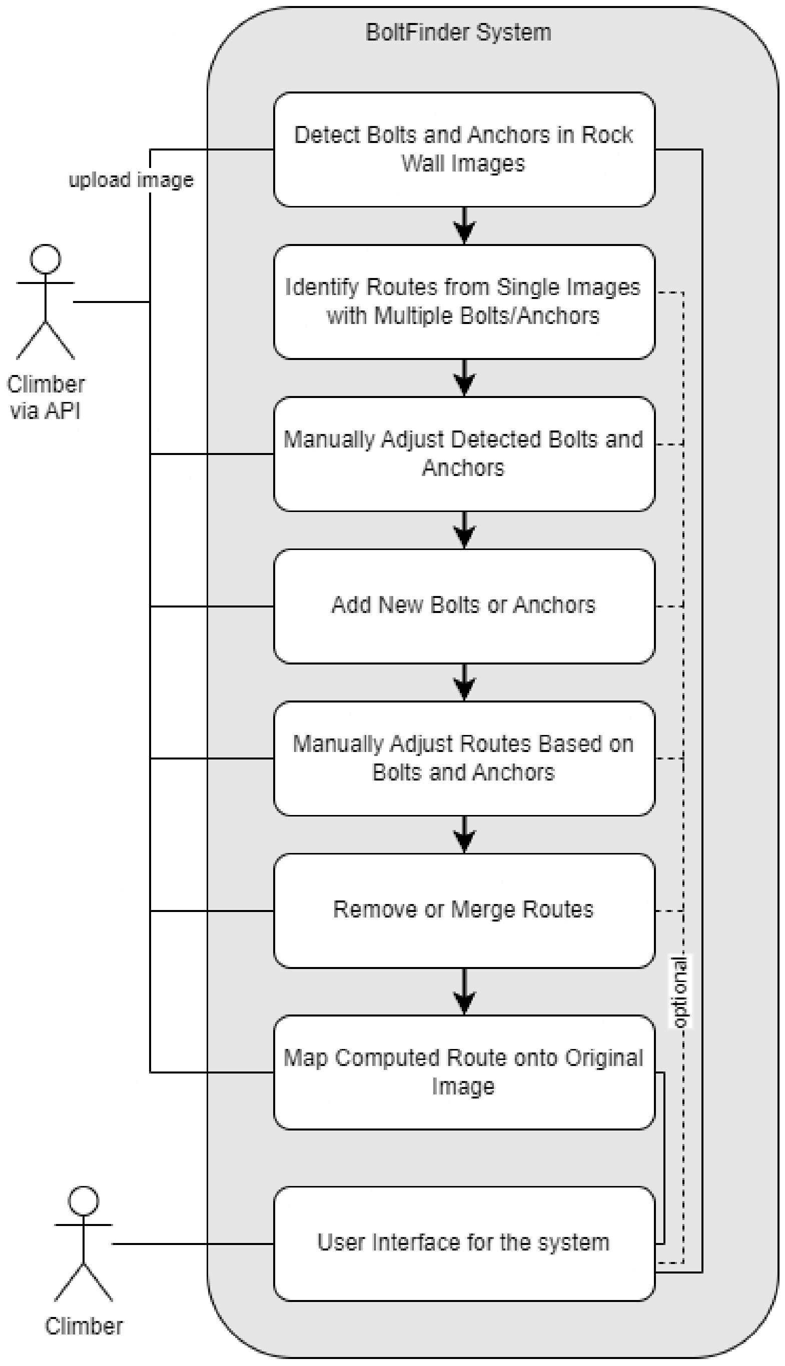

BoltFinder System
upload image
Climber via API
Climber
Detect Bolts and Anchors in Rock Wall Images
Identify Routes from Single Images with Multiple Bolts/Anchors
Manually Adjust Detected Bolts and Anchors
Add New Bolts or Anchors
Manually Adjust Routes Based on Bolts and Anchors
Remove or Merge Routes
Map Computed Route onto Original Image
User Interface for the system
optional

Figure 17: Use Case Diagram

II 2.4 Requirement Prioritization

The following terminology listed in Table 2 from RFC 2119 [20] is used to indicate the priority and requirement levels for the project. Each requirement is classified as Must, Should, or May, ensuring that essential functionalities are clearly defined and optional features are explicitly noted.

Priority	Description
Must	Requirements that are critical to the system's functionality. If not implemented, the system cannot fulfill its purpose. These are essential for the detection, pathfinding, and route mapping.
Should	Requirements that are important but not critical. Their absence might reduce the system's efficiency or user satisfaction, but the core functionality remains intact.
May	Optional requirements that enhance the system but are not necessary for its primary function. They provide additional features or improvements but can be omitted without major issues.

Table 2: Requirement Prioritization

II 2.5 Functional Requirements

The structure proposed by Craig Larman in his book "UML and Patterns" [21] is followed to document the use cases and functional requirements of the system. The following sections are omitted from the original structure for readability and clarity: "Main Success Scenario", "Extensions", and "Special Requirements". This approach ensures clarity and consistency in presenting the system's behavior, facilitating understanding for stakeholders and developers. The following tables describe the functional requirements of the system.

II 2.5.1 Detect Climbing Hooks in Rock Wall Images

Use Case Number	1
Priority	Must
Scope	The system under design (ML Application)
Level	User-goal
Primary Actor	System (ML Application)
Stakeholders and Interests	Climbers (require accurate bolt detection for route planning)
Preconditions	Mountain climbing images with visible bolts. The Bolts in the image must be minimum 30 pixels in either width or length and are at maximum 20% obscured by the wall or other objects.
Success Guarantee	A F1 score of 80% and recall of 90% for bolt predictions is achieved

Table 3: Functional Requirements: Bolt Detection

II 2.5.2 Generate a Graph Structure from Detected Hooks

Use Case Number	2
Priority	Must
Scope	The system under design (ML Application)
Level	Subfunction
Primary Actor	System (ML Application)
Stakeholders and Interests	Climbers (require the correct identification of individual routes)
Preconditions	The system has already detected bolts and anchors in an image with multiple routes
Success Guarantee	At least 60% of routes are identified correctly, 30% may have minor errors, and fewer than 10% are allowed to have major errors.

Table 4: Functional Requirements: Identify Routes from Single Images with Multiple Bolts/Anchors

II 2.5.3 Manually Adjust Detected Bolts and Anchors

Use Case Number	3
Priority	Must
Scope	The system under design
Level	User-goal
Primary Actor	Climber, System
Stakeholders and Interests	Climbers (want flexibility to correct any errors in detection)
Preconditions	The system has performed automatic detection of bolts and anchors
Success Guarantee	Users can manually adjust the placement of bolts and anchors in the image

Table 5: Functional Requirements: Manually Adjust Detected Bolts and Anchors

II 2.5.4 Add New Bolts or Anchors

Use Case Number	4
Priority	Must
Scope	The system under design
Level	Subfunction
Primary Actor	Climber
Stakeholders and Interests	Climbers (want flexibility to add missed bolts and anchors)
Preconditions	The system has completed initial detection, and the user needs to add undetected bolts or anchors
Success Guarantee	Users can add new bolts or anchors that are missed by the system's detection model

Table 6: Functional Requirements: Add New Bolts or Anchors

II 2.5.5 Manually Adjust Routes Based on Bolts and Anchors

Use Case Number	5
Priority	Must
Scope	The system under design
Level	User-goal
Primary Actor	Climber
Stakeholders and Interests	Climbers (need to modify routes if the system incorrectly assigns bolts and anchors)
Preconditions	The system has assigned bolts and anchors to routes
Success Guarantee	Users can manually reassign bolts and anchors to specific routes or create new routes

Table 7: Functional Requirements: Manually Adjust Routes Based on Bolts and Anchors

II 2.5.6 Remove or Merge Routes

Use Case Number	6
Priority	Must
Scope	The system under design
Level	Subfunction
Primary Actor	Climber
Stakeholders and Interests	Climbers (want flexibility to combine, remove routes or merge overlapping ones)
Preconditions	The system has assigned bolts and anchors to routes
Success Guarantee	Users can manually remove incorrect routes or merge routes that share common bolts and anchors

Table 8: Functional Requirements: Remove or Merge Routes

II 2.5.7 Map Computed Route onto Original Image

Use Case Number	7
Priority	Must
Scope	The system under design (ML Application)
Level	User-goal
Primary Actor	System (ML Application)
Stakeholders and Interests	Climbers (require visual representation of climbing routes)
Preconditions	The system has detected and assigned bolts and anchors to routes
Success Guarantee	The climbing routes are mapped back onto the original climbing wall image. The system generates a visual representation of the climbing routes by connecting the detected bolts and anchors with lines

Table 9: Functional Requirements: Map Computed Route onto Original Image

II 2.5.8 User Interface for the System

Use Case Number	8
Priority	May
Scope	The system under design
Level	User-goal
Primary Actor	Climber
Stakeholders and Interests	Climbers (require an intuitive interface for interacting with the system)
Preconditions	The system (API) is running and ready for user interaction
Success Guarantee	Users can interact with the system through a user-friendly interface to upload a image of a climbing wall with visible bolts and in return get the detected climbing routes drawn onto it

Table 10: Functional Requirements: User Interface for the system

The results regarding whether the Functional Requirements (FR) are achieved can be found in Chapter II 6.1.3.

II 2.6 Non Functional Requirements (NFR)

The following Non-Functional Requirements (NFRs) are defined based on the ISO/IEC 25010 [22] standard for system and software quality models. This ensures that the system adheres to high standards of performance, security, usability, and maintainability, among other quality attributes.

II 2.6.1 Time Behavior

Field	Description
ID	1
Category	Performance Efficiency
Subcategory	Time Behavior
Scenario	Processing images, detecting bolts, editing bolts and generating the final Topo
Trigger	The system must handle steps with minimal delays during processes
Reaction	The system ensures that in 90% of cases, the user is not left waiting without a notice of long running processes
Measure	A user uploads an image, edits data or generates the image and receives responses within 2s or is informed of a running process
Priority	Should
Status	Fulfilled

Table 11: Non-Functional Requirement: Time Behavior

II 2.6.2 User Error Protection

Field	Description
ID	2
Category	Interaction Capability
Subcategory	User Error Protection
Scenario	Users accidentally delete or misplace bolts or anchors during manual adjustments
Trigger	Manual UI interaction during bolt/anchor editing
Reaction	System prompts users with confirmation before making any destructive changes
Measure	destructive user actions are mitigated by confirmation prompts or undo functionality
Priority	May
Status	Fulfilled

Table 12: Non-Functional Requirement: User Error Protection

II 2.6.3 Modularity

Field	Description
ID	3
Category	Maintainability
Subcategory	Modularity
Scenario	Adding or updating components, such as new climbing site data or SAC integration
Trigger	Need for extending the system’s functionality without affecting core components
Reaction	System is built in a modular manner to allow independent updates to components without affecting the main detection system
Measure	Components are decoupled by API endpoints and can be updated independently
Priority	Should
Status	Fulfilled

Table 13: Non-Functional Requirement: Modularity

II 2.6.4 Integration with SAC

Field	Description
ID	4
Category	Compatibility
Subcategory	Interoperability
Scenario	Integrating BoltFinder system with SAC’s online platform
Trigger	The system must have interfaces for climbing route data, bolts, and anchors allowing integration with SAC’s portal
Reaction	Create API endpoints for the separate steps in the system. The data structure should be following the terminology used in the SAC portal but can deviate if needed.
Measure	Endpoints return defined data structures
Priority	Should
Status	Fulfilled

Table 14: Non-Functional Requirement: Integration with SAC (Schweizer Alpen-Club)

Detailed information on the status of the Non-Functional Requirements (NFR) can be found in the results Chapter II 6.1.3.

II 3 Analysis

This chapter provides an in-depth analysis of the current situation, similar or relevant model types, tool evaluation, and challenges. The analysis serves as the foundation for the subsequent development and realization of the project. For an overview of the State of the Art, refer to Chapter I 2 in the Technical Report.

II 3.1 Domain Model

This section provides an overview of the problem domain and the entities involved. The domain model in Figure 18 illustrates the relationships between the main entities and their attributes.

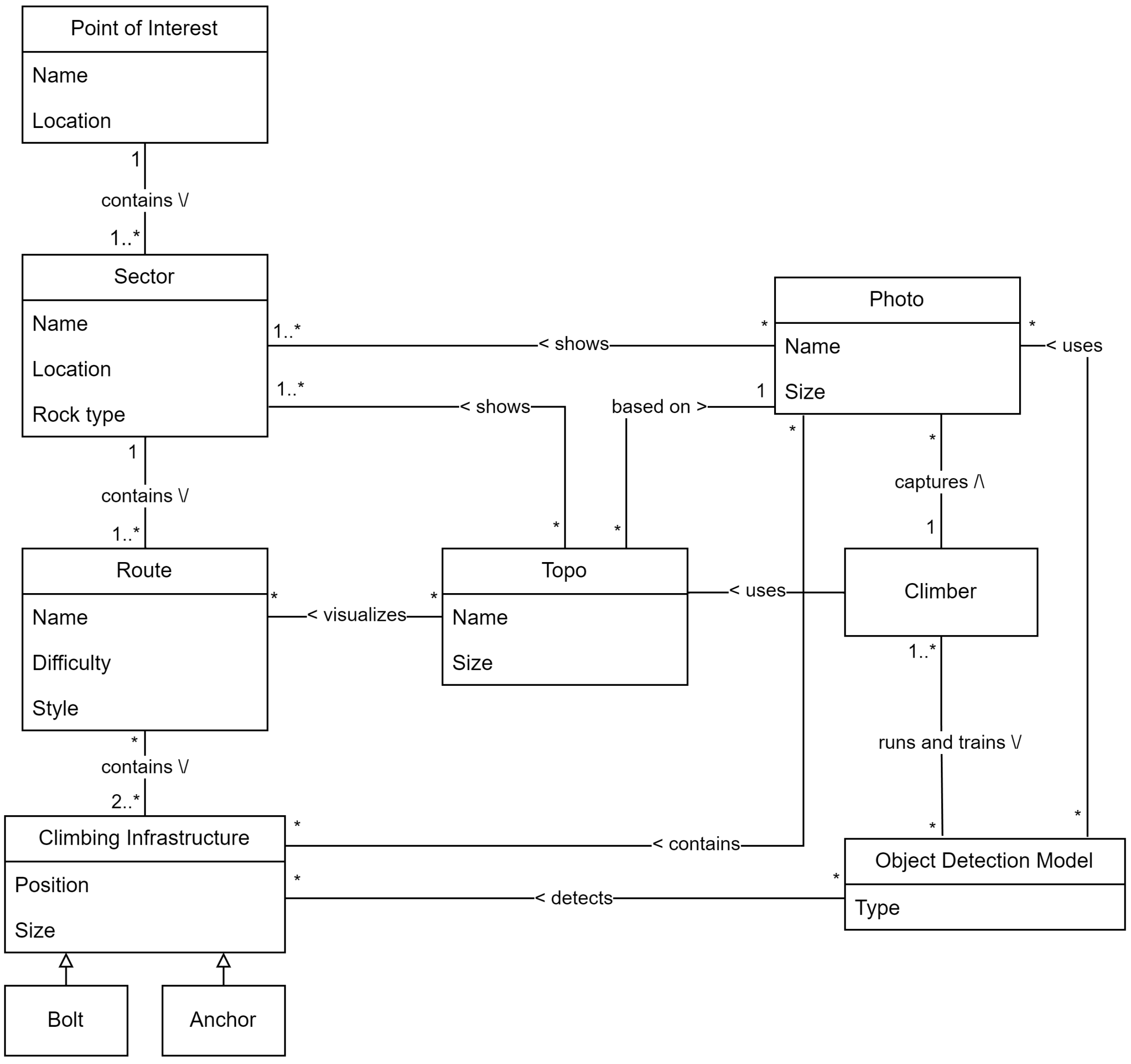


Figure 18: Domain Model

Note

The domain model is a high-level representation of this projects problem domain. It provides an overview of the entities and their relationships. The model is a simplified representation and may not capture all the details of the climbing domain.

II 3.2 Object Detection Models Research

Object detection models play a crucial role in identifying climbing bolts and anchors in rock wall images. The choice of model significantly impacts detection accuracy, speed, and scalability. This section explores the challenges associated with detecting small objects, evaluates popular object detection models and techniques, and explains the methods used to enhance detection performance for climbing routes.

II 3.2.1 Challenges in Detecting Small Objects

Detecting small objects, such as climbing bolts, presents unique challenges. These objects often occupy only a few pixels in large rock wall images, making them difficult to detect. Additionally, the variation in scale across images—from close-up shots to distant views—requires models that can effectively handle objects of varying sizes. Occlusions also pose a significant problem, as bolts may blend into the rock textures or be partially hidden by other elements. Furthermore, class imbalance is a pervasive issue, as datasets typically contain a significantly higher number of background pixels compared to climbing bolts. This imbalance often leads to uneven model learning, requiring specialized techniques such as slicing large images, dataset augmentation, and balanced loss functions to mitigate these challenges.

II 3.2.2 Evaluation of Detection Models

Several object detection models were evaluated for their ability to detect climbing bolts, focusing on their strengths and limitations within the context of this project. The assessment aimed to identify the model that best balances accuracy, recall, and efficiency while addressing the unique challenges of small object detection.

II 3.2.2.1 YOLOv11

YOLOv11 is identified as the most suitable model for this project due to its strong performance in small object detection. Its optimization for high recall, combined with real-time detection capabilities, makes it an ideal choice for detecting bolts and anchors. YOLOv11's adaptability, particularly through anchor box and scale tuning, further enhances its effectiveness for detecting small objects like climbing bolts. The comprehensive documentation available for YOLOv11 simplifies its training and customization, streamlining its deployment in real-world scenarios. In evaluations, YOLOv11 consistently outperforms other models in terms of both speed and accuracy, establishing itself as the primary model for this project. Further information on the YOLO model is available in Chapter II 5.2.2.1.

II 3.2.2.2 Faster R-CNN

Faster R-CNN is initially selected as a secondary model due to its precision and reliability in small object detection. Its region proposal mechanism provides accurate localization, making it well-suited for tasks requiring detailed post-processing. Despite its strengths in precision, Faster R-CNN produces weaker results compared to YOLOv11 in both accuracy and efficiency during evaluations. As a result, the decision is made to discontinue its use after the initial assessments in favor of focusing solely on YOLOv11. Further details on Faster R-CNN are provided in Chapter II 5.2.2.2.

II 3.2.2.3 Other Models

Several other models are considered based on their documented [23] capabilities but are ultimately excluded from detailed evaluation due to time constraints. RetinaNet, while noted for its effectiveness in addressing class imbalance with its focal loss function, is reported to struggle with the speed and small object detection performance required for this project. Transformer-based architectures such as DETR and DINO demonstrate innovative approaches but are described in the literature [24] as relying heavily on global context rather than localized proposals, which limits their ability to detect small objects like climbing bolts. Additionally, their high computational requirements and relatively limited documentation present practical challenges.

Detectron2, although versatile and widely used, offers functionality that overlaps significantly with Faster R-CNN. Given its added complexity and the absence of clear advantages for this specific task, it is not pursued further. Due to time limitations, only YOLOv11 and Faster R-CNN undergo detailed evaluation with actual trained models, as these are considered the most promising candidates for small object detection in climbing routes.

II 3.2.2.4 Final Model Selection

The evaluation highlights YOLOv11 as the most promising model for this project, excelling in both speed and accuracy for small object detection. While Faster R-CNN initially shows potential, its performance is ultimately surpassed by YOLOv11. As a result, the project proceeds with YOLOv11 as the sole model for further training and deployment. The detailed comparison between YOLOv11 and Faster R-CNN is discussed in Chapter II 6.1.1.1.

II 3.2.3 Detection Techniques

Enhancing detection performance for small objects requires a tailored approach. Large rock wall images are divided into smaller overlapping tiles using the SAHI library, ensuring that small objects are detected within manageable regions. Overlapping slices address edge cases where objects might be split across boundaries. Dataset augmentation plays a crucial role in improving model generalization by simulating various conditions through random cropping, resizing, flipping, and zooming. Synthetic data generation supplements the dataset with additional labeled samples, helping to counteract class imbalance.

High-resolution input during training and inference preserves the level of detail required for detecting small objects like bolts. Hard example mining prioritizes challenging cases during training, such as bolts partially obscured by textures or occlusions, improving the model's robustness. Weighted loss functions, such as focal loss, further address class imbalance by emphasizing the detection of bolts rather than background pixels. These techniques collectively ensure the models' ability to effectively handle the unique challenges of climbing bolt detection.

II 3.2.4 Role of the Dataset

The dataset plays a central role in enabling the object detection models to perform effectively. To improve diversity, augmentation techniques such as rotation, scaling, flipping, and color adjustments are applied. Synthetic data generation adds additional labeled samples of climbing bolts and anchors, further enhancing the dataset's quality. Class imbalance, a common issue in small object detection, is addressed by oversampling bolts and applying weighted loss functions like focal loss. These efforts result in a balanced and comprehensive dataset that allows the models to achieve robust performance in detecting climbing infrastructure.

II 3.3 Evaluation of Route Algorithms

Several algorithmic approaches are considered for generating climbing routes, each offering distinct strengths and trade-offs depending on the use case. Importantly, there are no precise or universal rules for creating climbing routes. Additionally, the goal of this project is not to find the shortest path but to identify all possible climbing routes based on detected bolts and anchors.

II 3.3.1 A*

The A* algorithm is a well-known graph traversal method that uses a heuristic to efficiently find the shortest path. It offers a high degree of customizability by allowing the heuristic to incorporate project-specific constraints, such as route preferences or difficulty levels. However, because the objective is to find all possible routes rather than a single optimal path, A* may require significant modifications to fulfill this purpose. Furthermore, it introduces moderate implementation complexity and may be computationally expensive for large datasets.

II 3.3.2 Dijkstra

Dijkstra's algorithm guarantees the shortest path in weighted graphs, making it robust and reliable for simple route generation. However, its primary focus on optimizing for a single shortest path limits its applicability to this project, which aims to identify multiple valid routes. While Dijkstra is simple and efficient for small graphs, its lack of customizability and scalability for large, complex climbing walls makes it less suitable for this use case.

II 3.3.3 Rule-Based

Rule-based methods define climbing routes using explicit, predefined rules. These systems are highly customizable, as constraints like allowable moves, maximum distances, and difficulty ratings can be encoded directly into the rules. They are also simple to implement and efficient, making them ideal for real-time applications. However, the lack of precise rules for climbing route generation makes a purely rule-based approach challenging to scale and adapt to diverse climbing scenarios. While rule-based methods can provide a strong foundation, they may need to be combined with other algorithms to ensure flexibility and adaptability.

II 3.3.4 Voronoi-Based

Voronoi-based algorithms partition the climbing wall into regions based on the proximity of climbing holds, creating a geometric representation of the wall. These methods excel in scalability, handling complex wall structures effectively, and can naturally generate multiple potential routes. However, they are less customizable than rule-based methods or A*, and their implementation involves moderate complexity, especially when integrating user preferences or climbing constraints. Despite these challenges, Voronoi-based approaches are well-suited to the goal of identifying all possible routes rather than optimizing for a single path.

II 3.3.5 Conclusion

After evaluating the possibilities, **rule-based** and **Voronoi-based** methods are selected as the primary algorithms for route generation. Rule-based methods offer simplicity, efficiency, and customizability, making them highly suitable for encoding general climbing guidelines. However, the absence of precise climbing rules means a purely rule-based approach may struggle with adaptability, particularly for non-standard scenarios. Voronoi-based methods complement this by providing a scalable and geometric approach that excels at generating multiple possible routes from the detected bolts and anchors. Together, these methods ensure a balanced solution that meets the project's objectives of flexibility, scalability, and ease of implementation.

Algorithms like A* and Dijkstra are not chosen as they are primarily designed for finding optimal or shortest paths, which is not the goal of this project. Additionally, their higher computational costs and limited adaptability make them less practical for generating multiple climbing routes.

II 3.4 Evaluation of Tools

This section showcases the tools used and considered for various aspects of the project, including image labeling, photogrammetry, and image stitching. Each tool is evaluated based on criteria such as functionality, ease of use, cost, and suitability for the project's specific requirements. The selected tools ensure a balance between efficiency, practicality, and scalability, aligning with the project's overall goals.

II 3.4.1 Image Labeling Tools

Image labeling tools are essential for annotating climbing infrastructure in images. Below is a comparison of local and web-based labeling tools.

II 3.4.1.1 Local Labeling Tools

Local tools run on personal machines, offering flexibility without requiring an internet connection. Table 15 compares three popular local labeling tools.

Criteria	LabelStudio (LabelImg)	LabelMe	QuPath
Runs on	localhost (slow)	localhost	localhost
Auto labeling	configurable	extensions	no
Export formats	*	COCO, JSON, YOLO (converter available)	CSV, JSON, XML
Price	free	free	free
Users	∞	∞	∞

Table 15: Local Labeling Tools Comparison

II 3.4.1.2 Web-Based Labeling Tools

Web-based tools provide additional features, such as cloud storage and auto-labeling, but may have limitations in free tiers. Table 16 compares four popular web-based labeling tools.

Criteria	Roboflow	SuperAnnotate	CVAT	Supervisely
Runs on	Web (image size limit)	Web	Web	Web
Cloud storage	yes	yes	yes	yes
Auto labeling	paid/academic?	paid	500 requests/day	yes
Export formats	*	*	*	*
Price	academic tier (review and citation)	free-tier	free-tier	free-tier (5GB storage limit)
Users	3	2	2	2

Table 16: Web-Based Labeling Tools Comparison

Note: "*"" indicates support for modern formats like COCO, VOC, and YOLO.

II 3.4.1.3 Tool Selection and Justification

LabelMe is selected for this project due to its:

- Free and open-source nature.
- Local operation, which supports collaboration without internet dependencies.
- Support for multiple export formats.

Web-based tools like **Roboflow** and **SuperAnnotate** offer advanced features like cloud storage and auto-labeling but have limitations, such as file size or user restrictions in their free tiers, making them less suitable.

II 3.4.2 Photogrammetry Tools

Photogrammetry tools allow for creating 3D models of climbing walls, enhancing data visualization. The following tools are considered:

- **Alice Vision Meshroom**: Open-source and suitable for generating 3D models.
- **Open-SFM**: Focuses on large-scale image reconstruction.
- **Napari**: An interactive viewer for n-dimensional data.
- **COLMAP**: A popular tool for photogrammetry tasks.
- **Polycam**: A user-friendly option for quick 3D scans.

Conclusion: Alice Vision Meshroom identifies as the most compatible tool due to its open-source nature and robust feature set. However, photogrammetry is not included in this project. This decision reflects the higher cost and limited availability of the hardware required for 3D scanning compared to standard cameras. By focusing on camera-based methods, the BoltFinder approach remains practical and scalable, enabling its application across climbing locations throughout Switzerland.

II 3.4.3 Image stitching tools

Image stitching tools combine multiple images into a single panoramic view. These tools are invaluable for photography, mapping, and visualization projects. Below in Table 17 is a comparison of three popular tools for image stitching:

Criteria	**Microsoft Image Composite Editor** [25]	**Adobe Photoshop/Lightroom** [25]	**Hugin** [26]
Cost	Free	Paid (Subscription)	Free
Automatic image ordering	yes	yes	no
Ease of use	High	Medium (requires expertise)	Medium (manual steps)
Output quality	High	Very high	High
Customizability	Low	High	Very high
Supported formats	Various	Extensive	Extensive
Reason for choice	Familiarity and experience with the tool	Not chosen due to cost	Not chosen due to manual complexity

Table 17: Image stitching tools comparison

Microsoft ICE is used for this project, because of previous experience with the tool, making it an ideal choice for quick and efficient panoramic stitching. Adobe Photoshop and Hugin are also excellent tools but are not chosen due to cost or manual complexity, respectively. The choice of tool depends on the user's familiarity, budget, and specific project requirements.

The stitched images are visualized in Chapter II 5.1.7.

II 3.5 Evaluation of Technologies

This section provides an overview of the key technologies used in the project, focusing on their roles and advantages in addressing specific requirements. Each technology is chosen based on its scalability, ease of use, and suitability for the project's diverse needs, from inference APIs to frontend development, database management, and model tracking.

II 3.5.1 FastAPI

FastAPI is chosen for the inference API due to its high scalability, ease of use, and exceptional performance in handling asynchronous HTTP requests. It is particularly well-suited for deploying machine learning models in production environments, where latency is critical. FastAPI also provides excellent integration with Python, which is the primary language used for this project's model training and inference tasks. Its active and extensive community further ensures long-term support.

II 3.5.2 Express

Express is selected for the backend API due to its flexibility and wide adoption in web application development. As a lightweight framework for Node.js, it provides high performance and is scalable for handling complex routing and middleware. While its ease of integration is moderate compared to FastAPI, its extensive community and ecosystem of plugins make it a reliable choice for this project.

II 3.5.3 React

For the frontend, React is chosen due to its scalability and ease of integration. React's component-based architecture simplifies the development of a responsive and dynamic user interface, which is essential for the project's visualization and user interaction needs. Its large community and robust ecosystem of libraries make it an ideal choice for modern web development.

II 3.5.4 PostgreSQL

PostgreSQL is selected as the database solution for its high scalability and robust support for complex queries. Its performance and reliability make it well-suited for managing the structured data generated during the analysis and pathfinding stages of the project. While its ease of integration is slightly more involved compared to NoSQL solutions, its extensive community ensures that support and resources are readily available.

II 3.5.5 Cantaloupe

Cantaloupe is chosen as the image server for its optimization in serving and processing large images. Its scalability ensures that high-resolution rock wall images can be efficiently served to both the backend and frontend systems. Although its community support is less extensive than some other frameworks, it provides the performance and specialized functionality needed for this project.

II 3.5.6 Jupyter Notebooks

Jupyter Notebooks are selected for model training due to their interactive and user-friendly environment. They enable rapid prototyping and experimentation, which is essential during the development and fine-tuning of the object detection models. While not optimized for high-performance training on large datasets, their extensive community and integration with Python make them a practical choice for the research and development phase.

II 3.5.7 MLflow

MLflow is chosen as the model registry for its comprehensive tracking and management of machine learning models. It offers high scalability, allowing models to be versioned, tracked, and deployed efficiently. Its ease of integration with Python and other machine learning frameworks, combined with its strong community support, makes it a reliable choice for production-grade machine learning workflows.

II 3.6 Conclusion

The selected frameworks collectively address the diverse requirements of this project, ensuring scalability, performance, and ease of integration. FastAPI and Express provide robust solutions for the inference and backend APIs, respectively, while React enables a responsive and dynamic user interface. PostgreSQL ensures reliable management of structured data, and Cantaloupe is optimized for serving large rock wall images. For model development and deployment, Jupyter Notebooks facilitate rapid prototyping, while MLflow provides comprehensive tools for model tracking and registry. The combination of these frameworks ensures a seamless pipeline from model training to deployment, supporting the core objectives of the project.

II 4 Design

This chapter describes conceptual and design aspects of the project. It includes the architecture, user interface design, and deployment strategies. Implementation details are found in the next Chapter II 5.

II 4.1 System Architecture Overview

This section provides a high-level overview of the BoltFinder system and its components.

II 4.1.1 Purpose

- Illustrate how major components interact.
- Provide context for detailed design subchapters.

II 4.1.2 High-Level Design

The following Figure 19 C4 diagram depicts the system architecture of the BoltFinder project. The diagram outlines the key components and their interactions, highlighting the system's structure and the flow of data and control.

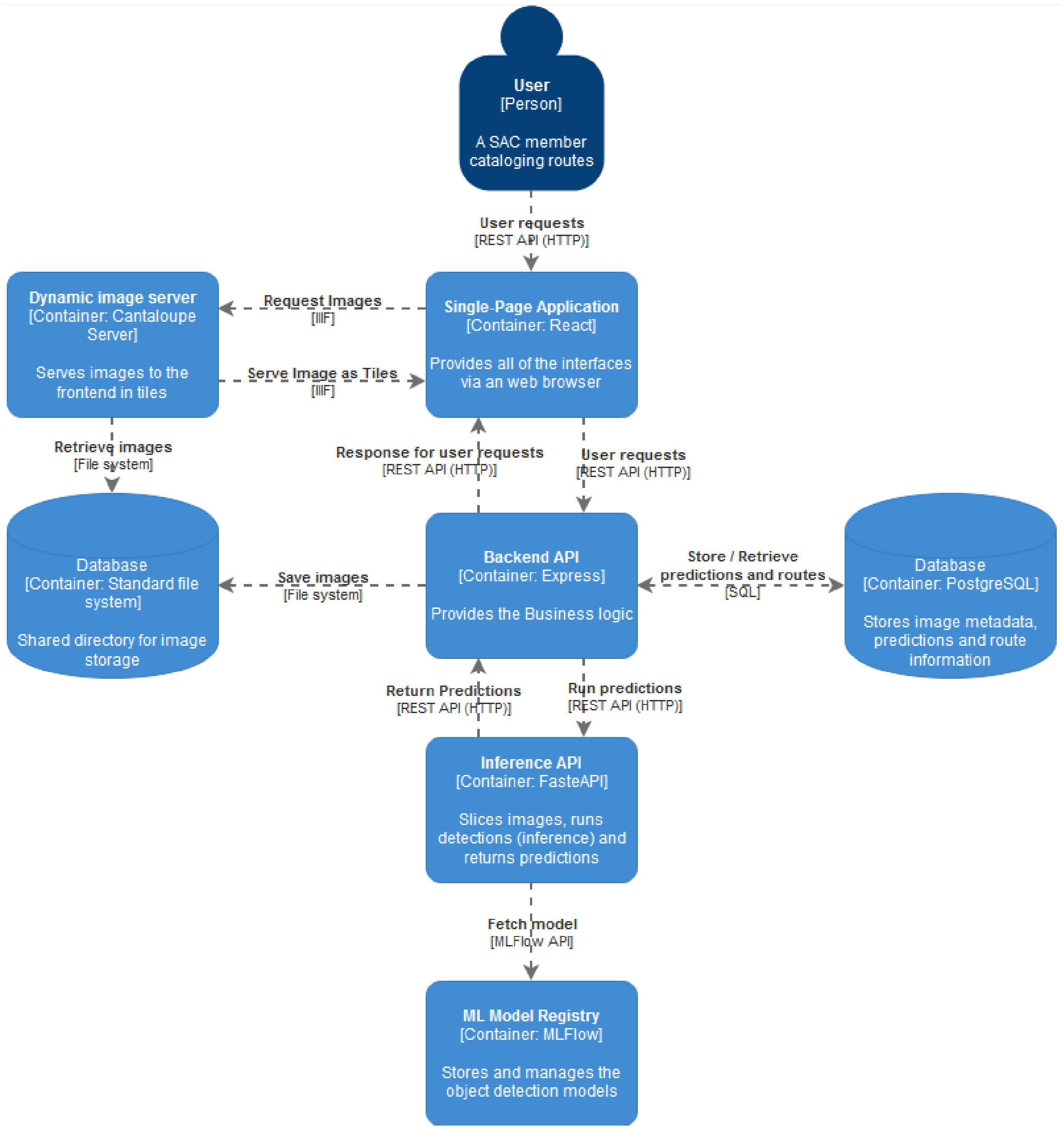


Figure 19: C4 Container Diagram of the BoltFinder Project.

II 4.1.2.1 Key Components

1. **Inference API**: Handles object detection requests.
2. **Backend API**: Coordinates workflows, stores predictions, and generates climbing routes.
3. **Frontend**: User-facing interface for uploads, predictions, and route visualization.
4. **Database**: Stores images, predictions, and generated routes.
5. **Cantaloupe Server**: Provides a mechanism for serving images uploaded via the frontend.

II 4.1.3 Dedicated Subsystems

The BoltFinder system is divided into distinct subsystems to manage specific responsibilities effectively and independently.

II 4.1.3.1 Inference API and MLFlow Communication

The Inference API is the only component allowed to communicate with the MLFlow server. This design encapsulates all model management operations, including fetching, versioning, and caching, within a single subsystem. By centralizing this interaction, the system enforces security and reduces complexity, ensuring that other components do not need to implement or manage model-related logic.

II 4.1.3.2 Cantaloupe Server and Image Serving

The Cantaloupe Server provides a crucial role in the system by enabling real-time access to images stored on the server. When images are uploaded via the frontend, they are stored in a directory accessible to both the Cantaloupe Server and the backend API. This architecture allows the frontend to directly retrieve images for visualization or validation purposes without requiring additional processing from the backend.

Many uploaded images are exceptionally large, with sizes nearing 1GB. The Cantaloupe Server addresses this challenge by serving images in tiles, allowing only the necessary portions of the image to be loaded based on the user's viewport. As the user zooms in, additional image details are dynamically retrieved.

Implications of this design include:
- **Efficiency**: Direct access to stored images reduces latency for image retrieval tasks. Serving images in tiles ensures that only the required portions are loaded, optimizing bandwidth usage.
- **Scalability**: This approach supports high-throughput scenarios by offloading image serving responsibilities to the Cantaloupe Server, allowing the backend API to focus on computation-heavy tasks.

II 4.1.3.3 Separation of Inference and Backend Logic

The decision to separate the Inference API from the backend API ensures a clear distinction between business logic and object detection logic. This design choice has several benefits:

- **Modularity**: Each subsystem can focus on its specific responsibilities. The backend API handles business workflows, such as managing user interactions, storing predictions, and generating climbing routes, while the Inference API is dedicated to processing object detection requests.
- **Scalability**: By decoupling the two subsystems, the Inference API can scale independently to handle high volumes of detection requests without impacting the performance of the backend API.
- **Maintainability**: Separation simplifies debugging and maintenance. Changes to object detection models or logic in the Inference API do not risk affecting the business workflows managed by the backend API.
- **Technology Flexibility**: The backend API and Inference API can use different technology stacks optimized for their respective functions, ensuring better performance and efficiency across the system.

This architectural decision underscores the importance of maintaining clear boundaries between subsystems to enhance the overall robustness and flexibility of the BoltFinder system.

II 4.1.4 High Level Data Flow

The following diagram in Figure 20 illustrates the high-level data flow within the BoltFinder system. It outlines the sequence of operations from image upload to route generation to the final export of the topo image, highlighting the interactions between the key components. The detailed implementations and interactions between the components are described later in Chapter II 5.

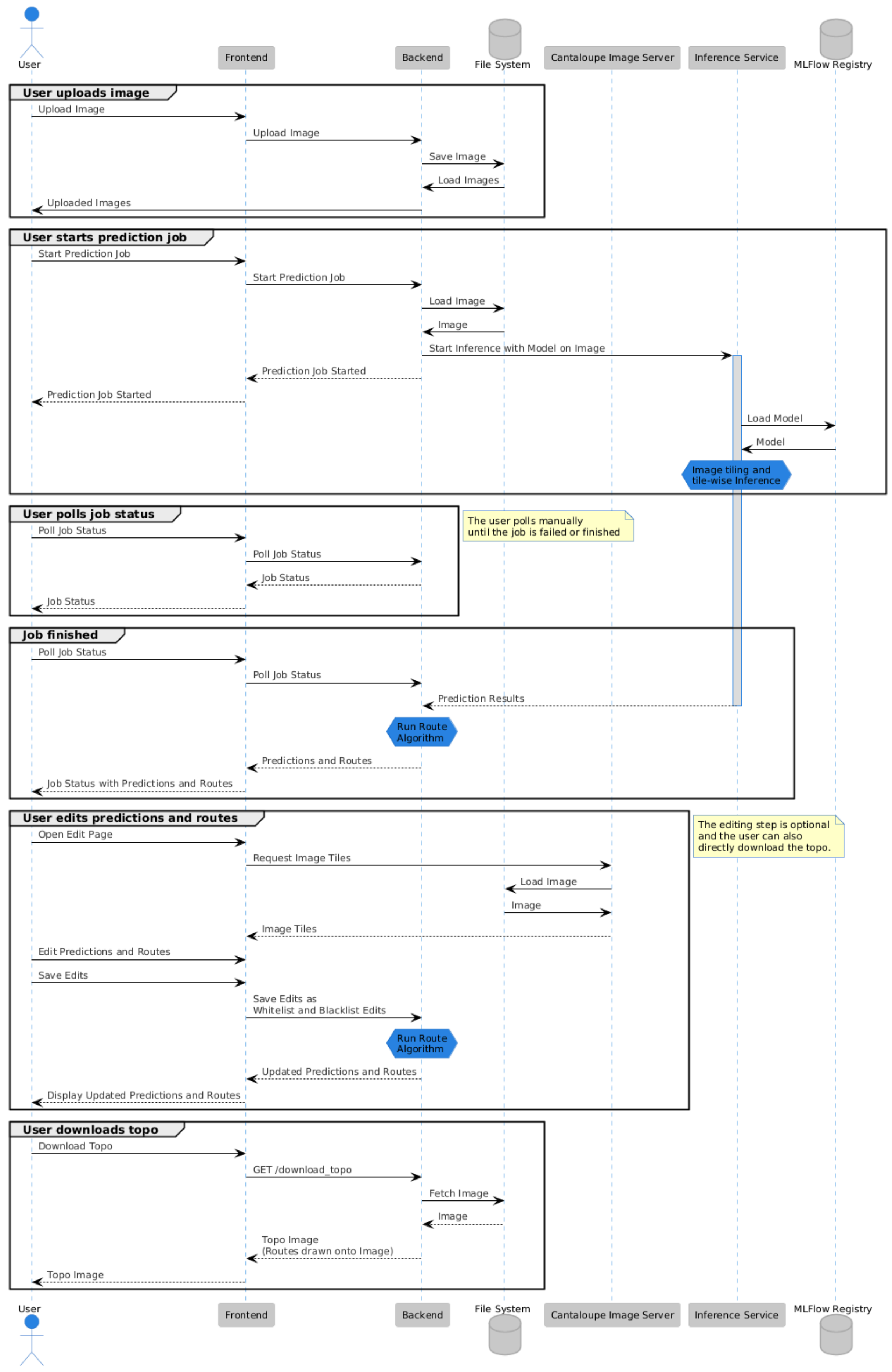


Figure 20: High-level Data Flow Diagram of the BoltFinder System.

II 4.2 Inference API Design

This section explores the architecture and design considerations of the Inference API, a critical component of the BoltFinder system. It focuses on enabling efficient, scalable, and extensible inference workflows for object detection tasks.

II 4.2.1 Purpose

The Inference API bridges the gap between users and the object detection models, providing RESTful endpoints for listing available models and running predictions. Designed with scalability in mind, it ensures seamless integration of new models while automating processes like model fetching and caching through MLFlow. The Figure 21 illustrates the high-level architecture of the Inference API.

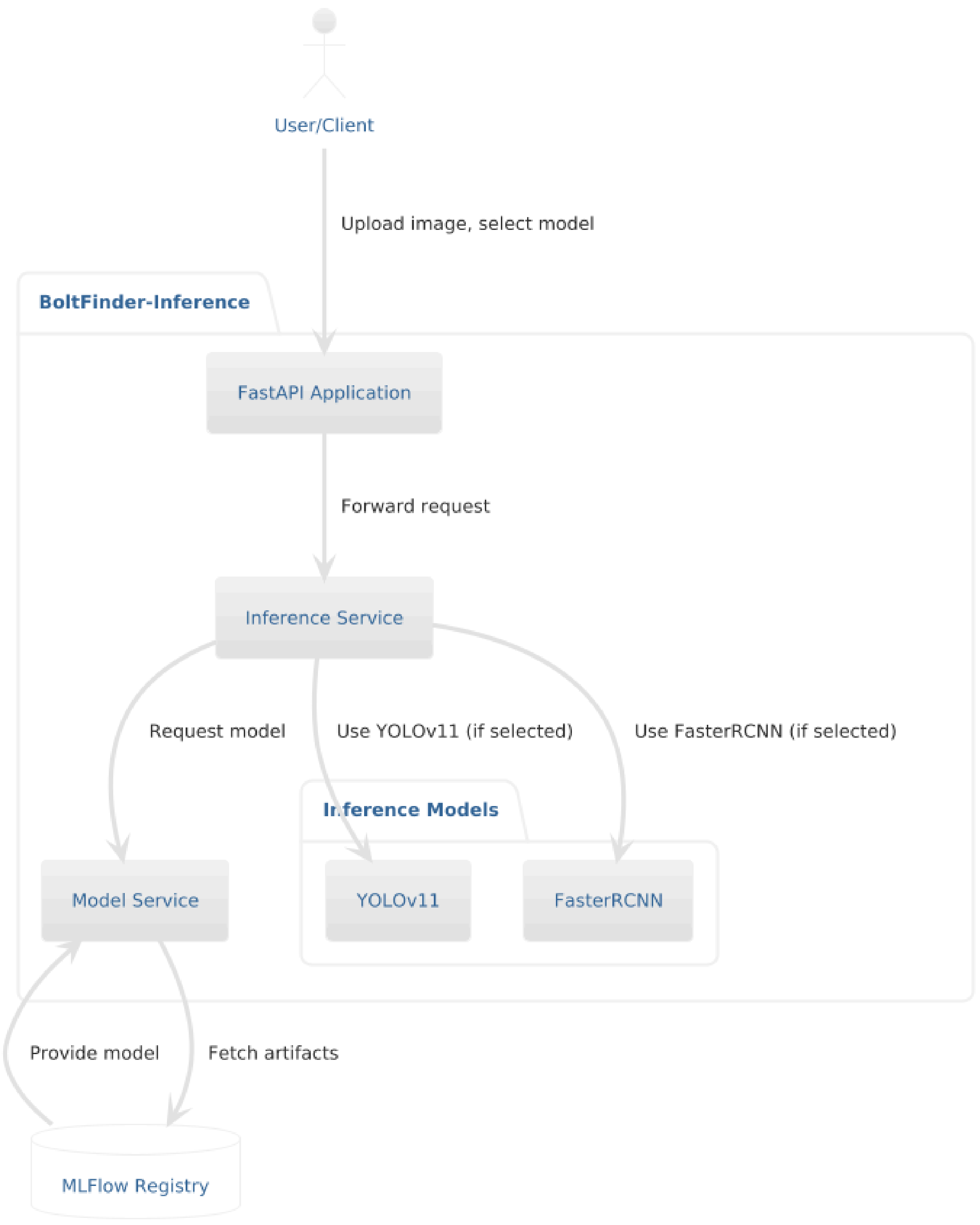


Figure 21: High-level architecture of the BoltFinder-Inference subsystem.

II 4.2.2 Architecture

The Inference API architecture adopts a layered approach that isolates key functionalities into distinct components. This modularity ensures maintainability and scalability as the system evolves.

- The **API Layer** exposes endpoints and handles client interactions.
- The **Inference Service** orchestrates workflows, including preprocessing, inference execution, and postprocessing.
- The **Model Service** integrates with MLFlow for dynamic listing, fetching and caching of model artifacts.
- **Utilities** provide essential support for tasks like image slicing and file management.

II 4.2.3 Extensibility

One of the core strengths of the Inference API is its extensibility. By following a standardized interface, adding support for new models requires minimal effort. Developers implement the required methods, such as `load_model` and `run_inference`, within the defined interface. This abstraction decouples model-specific details from the core workflow, enabling seamless integration.

II 4.2.4 Workflow and Caching

Upon receiving an inference request, the API initiates a structured pipeline. Images are preprocessed—often sliced into smaller tiles to enhance small-object detection—and the appropriate model is fetched. Cached models are used when available; otherwise, the system retrieves and stores the model from MLFlow dynamically. Predictions are refined using postprocessing techniques like non-maximum suppression (NMS).

II 4.2.5 Key Advantages

The design achieves several objectives:
- **Modularity**: Separate components allow for independent updates and testing.
- **Efficiency**: Cached models and preprocessing optimizations reduce latency and enhance performance.
- **Scalability**: Support for new models and workflows ensures adaptability to future requirements.
- **Automation**: Integration with MLFlow automates model lifecycle management, minimizing manual intervention.

The Inference API is a robust foundation for handling diverse object detection tasks while remaining flexible and scalable for future enhancements.

II 4.3 Backend API Design

II 4.3.1 Purpose

The backend API is the core of the BoltFinder system, handling workflows for job creation, predictions, route generation, and topology management. It serves as a bridge between the user-facing frontend and the system's internal logic.

II 4.3.2 High-Level Architecture

The backend API is built on a modular design, dividing functionality into controllers, services, and entities. This structure ensures maintainability, scalability, and clear separation of concerns. The following Figure 22 figure illustrates the high-level architecture of the backend API.

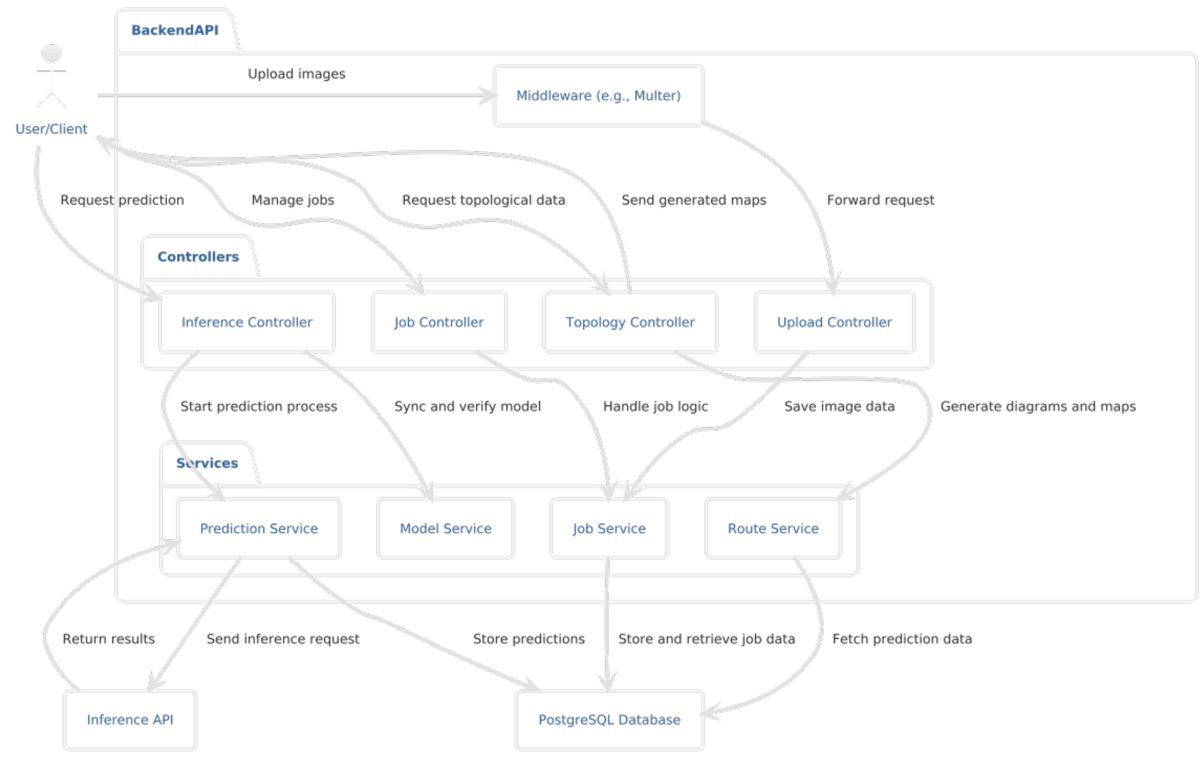


Figure 22: High-level architecture of the Backend API.

Key Components:
1. **Controllers**: Handle incoming HTTP requests and route them to the appropriate services.
2. **Services**: Contain business logic, including workflows for job creation, predictions, and route generation.
3. **Entities**: Represent database models, with repositories managing database interactions.
4. **Middleware**: Validates requests and handles errors globally.
5. **Utilities**: Provide helper functions for tasks like logging, configuration, and visualization.

Integration with the Inference API:
- The backend API communicates with the Inference API for object detection tasks, decoupling detection logic from business workflows.

Central Role of Jobs: The concept of a **job** serves as the backbone of the backend API, tying together all workflows and components. A job is a cohesive unit that encapsulates:
- The uploaded image to be processed.
- The prediction options selected, such as slicing parameters or confidence thresholds.
- The model chosen for inference.
- The current status of the job (e.g., pending, in progress, or completed).
- The predictions returned by the Inference API, stored as part of the job.

This abstraction ensures a unified workflow for the entire system, enabling seamless tracking of tasks and their outcomes.

Data Flow:
1. Users interact with the API through endpoints.
2. Controllers validate input and delegate tasks to services.
3. Services interact with repositories and utilities to fulfill requests.
4. Responses are returned to the client or processed further.

II 4.4 Frontend Design

This section provides an overview of the architecture and functionality of the frontend, highlighting its modular design and integration with system services to deliver an intuitive and efficient user experience.

II 4.4.1 Purpose

The frontend is designed to serve as the primary user interface for interacting with the system. Its modular architecture ensures scalability, maintainability, and seamless communication with backend services.

II 4.4.2 Architecture

The frontend employs a component-based architecture using React and TypeScript. It integrates with key services, including:

- **Backend API**: Handles image uploads, retrieves prediction data, and stores user modifications.
- **Inference API**: Runs object detection models for real-time predictions and annotations.
- **Cantaloupe Server**: Manages high-resolution, zoomable image tiles for detailed visualization.

The architecture emphasizes performance and responsiveness, leveraging modern tools to handle large datasets and ensure a smooth user experience across devices. The Figure 23 figure illustrates the core components and their interactions.

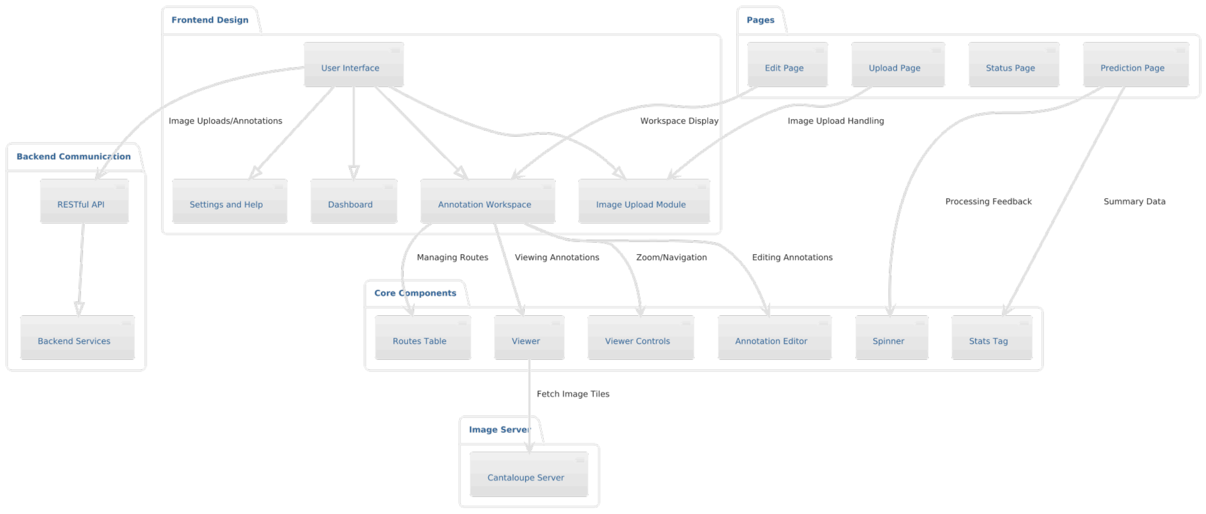


Figure 23: Frontend Component Diagram

II 4.4.3 Components

The frontend is structured around the following core components:

- **Upload Manager**: Facilitates image uploads with progress tracking and error handling. It communicates with the backend to ensure data integrity.
- **Prediction Viewer**: Displays object detection results on climbing wall images, utilizing the Cantaloupe Server for smooth rendering and navigation. It allows users to toggle annotation layers and customize the visualization.
- **Route Editor**: Provides tools for modifying detected routes and anchors. It supports precise adjustments with drag-and-drop functionality and validates changes in real time.

These components work together to create an intuitive workflow, guiding users from image uploads to route editing and finalizing predictions.

II 4.4.4 User Workflow

The diagram below in Figure 24 illustrates the primary user workflow:

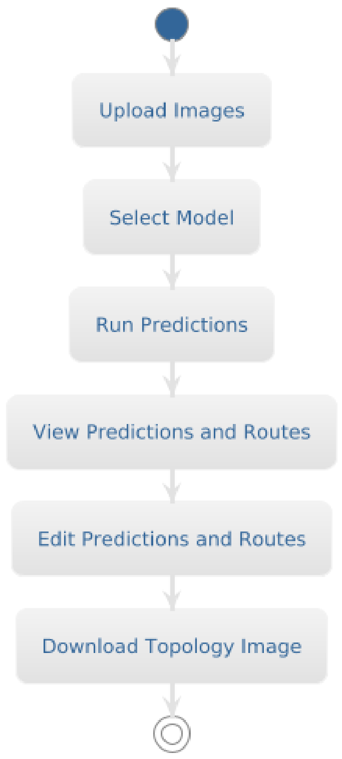


Figure 24: Frontend Workflow Diagram

1. **Upload Images**: Users begin by uploading high-resolution climbing wall images.
2. **Select Model and Run Predictions**: Users choose an appropriate detection model to analyze the images and generate predictions.
3. **View Predictions and Routes**: Predictions are visualized as overlays on the images, with detected routes and anchors displayed.
4. **Edit Predictions and Routes**: Users refine the predictions and routes using intuitive editing tools.
5. **Download Topology Image**: Users can download an Image with the bolts, anchors and Routes visible.

II 4.4.5 Additional Details

The frontend's responsive design ensures compatibility across devices, from desktops to field-use tablets. Key optimizations include:

- Lazy loading of components and data to improve performance.
- Memoization techniques to reduce redundant computations.
- Comprehensive error handling to enhance reliability.

Mockups showcasing the interface layout and workflows can be found in Appendix A Chapter III 2.

II 4.5 Model Training Design

The object detection training process integrates two complementary models, Faster R-CNN and YOLO, in a well-structured workflow. This workflow spans dataset preparation, model training, evaluation, and deployment, emphasizing robust architecture and efficient design.

II 4.5.1 Overview of the Workflow

The training workflow shown in Figure 25 begins with dataset preparation, where raw images and annotations in YOLO format are converted into the COCO format required for Faster R-CNN. The dataset is split into training and validation subsets to ensure balanced evaluation. Once prepared, the dataset serves as the foundation for training either Faster R-CNN or YOLO models, depending on the application needs.

Faster R-CNN utilizes a ResNet-50 backbone with a Feature Pyramid Network (FPN) for multi-scale feature extraction and a Region Proposal Network (RPN) to identify areas of interest. In contrast, YOLO employs a single-stage detection architecture optimized for real-time inference, predicting bounding boxes and class probabilities directly.

Both models undergo evaluation using COCO metrics such as mean Average Precision (mAP) and F1 scores. After validation, the trained models are saved for deployment. Both are logged with MLFlow for reproducibility.

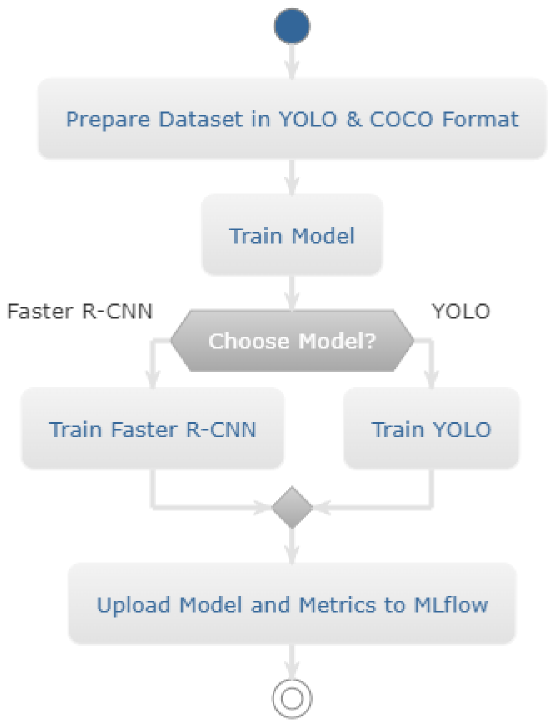

Prepare Dataset in YOLO & COCO Format
Train Model
Faster R-CNN
Choose Model?
YOLO
Train Faster R-CNN
Train YOLO
Upload Model and Metrics to MLflow

Figure 25: High-level workflow for object detection training, covering dataset preparation, model training, and deployment.

II 4.5.2 Summary

This workflow is designed to streamline the process of generating new datasets and training models. By supporting both YOLO and COCO formats, the workflow allows seamless dataset preparation, making it adaptable for diverse object detection tasks. The clear structure of the training process ensures that new models can be trained efficiently with minimal setup, enabling rapid iterations and scalability for future applications. The model training process is further detailed in the following sections Chapter II 5.1.5, Chapter II 5.2.1.

II 4.6 Deployment Design Overview

This section provides a brief overview of how BoltFinder components are deployed using a containerized approach with Docker Compose.

II 4.6.1 Purpose

The deployment aims to ensure seamless integration and scalability of all components through containerized services, enabling efficient development, testing, and production workflows.

II 4.6.2 Deployment Architecture

- The architecture consists of multiple containerized services:
  - **Frontend**: Handles the user interface, served as a static site or through a development server.
  - **Backend API**: Processes application logic and provides RESTful endpoints.
  - **Inference API**: Runs machine learning inference tasks, leveraging GPU acceleration via NVIDIA drivers.
  - **Database**: A PostgreSQL container for persistent application data storage.
  - **Image Server**: A Cantaloupe server for serving processed image data.

II 4.6.3 Docker Compose Setup

The deployment leverages Docker Compose to manage the services. Key features include:
- Unified configuration of ports, environments, and volumes for consistency.
- Persistent storage for database and file data via Docker volumes.
- GPU support for the inference service using NVIDIA Docker runtime.

II 4.6.4 Summary

This deployment design ensures that all BoltFinder components are containerized, scalable, and maintainable. Each service operates independently while remaining interconnected within a unified ecosystem. The infrastructure is further described in Chapter II 5.6.

II 5 Implementation and Testing

The following sections describe the implementation details of the software components and the testing of the software.

II 5.1 Dataset

This chapter explains the labelling and creation of the datasets used for training as well as testing the object detection models. The datasets can be defined in multiple stages:

- **Base dataset**, contains all the images and annotations.
- **Training datasets**, multiple versions of the base dataset which used only parts of the base dataset and processes them in different ways.
- **Testing datasets**, the panorama images which are used to test the models.

II 5.1.1 Dataset Challenges

The dataset creation is challenging due to the following reasons:

- slicing the images into tiles without having annotations that are cut off.
- scaling down images with high resolution.
- labeling of images.
  - finding small and blurry bolts and anchors.
  - labeling obscured bolts and anchors.
- manage different versions and slices of the dataset.

II 5.1.2 Dataset Formats

The datasets are stored in the following formats:

- LabelMe (per file json).
  - used for labeling the base dataset.
- YOLO (per file .txt and overall dataset.yaml).
  - Converted from the LabelMe format.
- COCO (overall json).
  - The COCO format is used for the Faster R-CNN model.
  - Converted from the YOLO format.

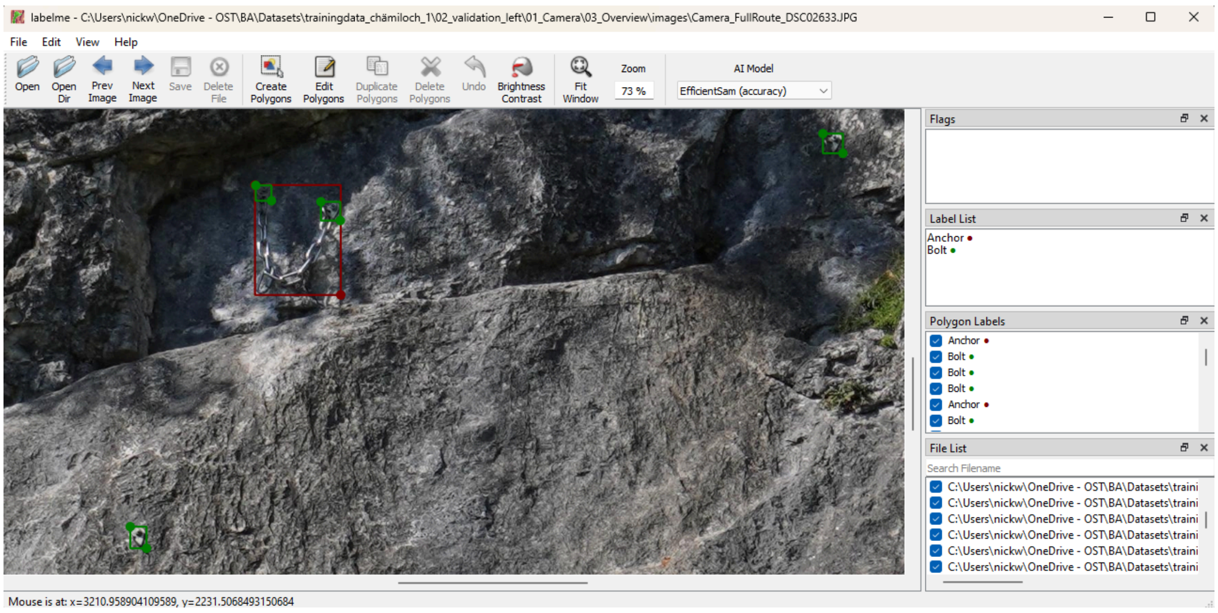
II 5.1.3 Labeling the Base Dataset

The base dataset is labeled using the LabelMe annotation tool (described in Chapter II 3.4.1). The images are gathered and labeled as part of this project. The images are captured from the climbing sector “Chämiloch” in Schwyz, Tessin and some anchors from the Web.

Two classes are annotated with rectangular bounding boxes:
- Bolts
- Anchors

The labeling is done with these rules in mind:
- mark all kind of bolts and anchors.
  - rusty and old ones.
  - bolts that are inside of anchors.
  - bolts that are inside of holes.
  - partly visible bolts and anchors.
  - blurry bolts and anchors.
- labels should fit tightly around the bolt or anchor.
- label only the visible part of the bolt or anchor.

In the screenshot (Figure 26) below the LabelMe tool is shown with some labeled bolts and anchors. And as described two bolts are inside of an anchor.


labelme - C:\Users\nickw\OneDrive - OST\BA\Datasets\trainingdata_chämiloch_1\02_validation_left\01_Camera\03_Overview\images\Camera_FullRoute_DSC02633.JPG

Zoom: 73 %

AI Model: EfficientSam (accuracy)

Label List: Anchor ●, Bolt ●

Polygon Labels: Anchor ●, Bolt ●, Bolt ●, Bolt ●, Anchor ●, Bolt ●

File List:
C:\Users\nickw\OneDrive - OST\BA\Datasets\traini
C:\Users\nickw\OneDrive - OST\BA\Datasets\traini
C:\Users\nickw\OneDrive - OST\BA\Datasets\traini
C:\Users\nickw\OneDrive - OST\BA\Datasets\traini
C:\Users\nickw\OneDrive - OST\BA\Datasets\traini
C:\Users\nickw\OneDrive - OST\BA\Datasets\traini

Mouse is at: x=3210.958904109589, y=2231.5068493150684

Figure 26: LabelMe tool with labeled bolts and anchors

The following Figure 27 shows a sample of the labeled bolts from the panorama images. The annotations follow the rules described above like keeping the annotation tight around the bolt and labeling bolts in anchors.

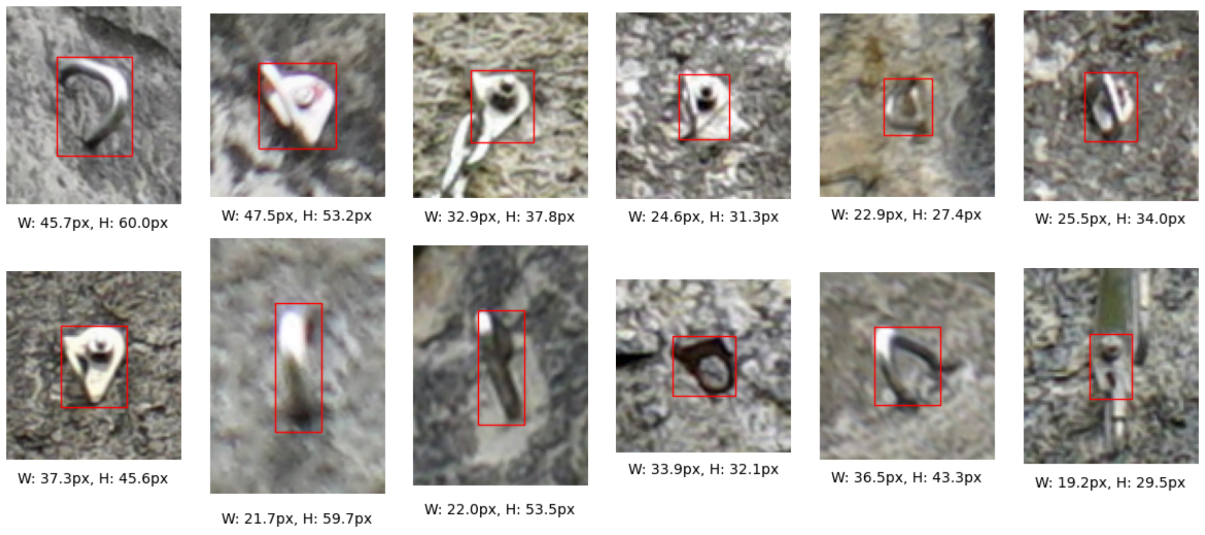


Figure 27: Labeled bolts from the panorama images

The same is shown for anchors in the following Figure 28.

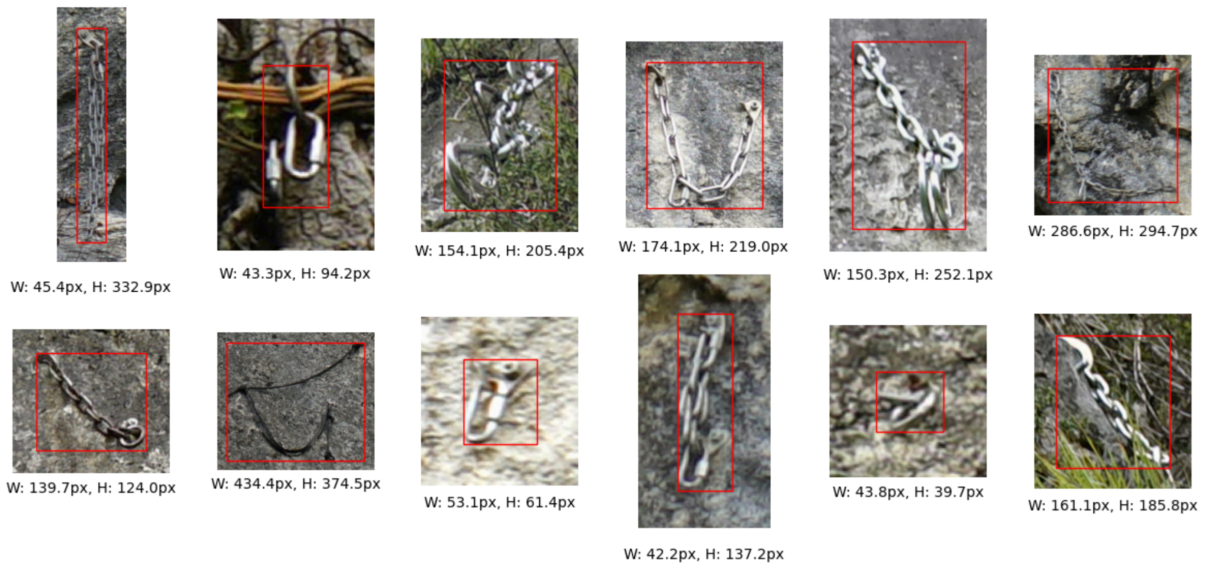


Figure 28: Labeled anchors from the panorama images

II 5.1.4 Base Dataset Overview

The base dataset consists of 832 images which contain 3'995 bolts and 269 anchors.

The dataset is divided into the train and test folder and they contain the folders:
Camera, Phone, Panorama, Web, Tessin, and Backgrounds.

The folders are further split into subfolders: close-ups, medium, and overview.
- The close-ups contain on average 1 annotation.
- The medium images contain on average multiple annotations and are further away.
- The overview images contain lots of annotations and are even further away from the wall.

II 5.1.4.1 Training & Validation Folder

The camera, phone and panorama folders only contain images from the right sector of the climbing sector "Chämiloch" in Schwyz. The following structure is used for the training folder:

- Camera
  - close ups
  - medium
  - overview
- Phone
  - close ups
  - medium
  - overview
- Panorama
  - overview
- Web
  - close ups
  - medium
- Tessin
  - close ups
  - medium
- Backgrounds
  - medium

In total it contains 566 images and 2238 bolts and 126 anchors as shown in Figure 29 and Figure 30.

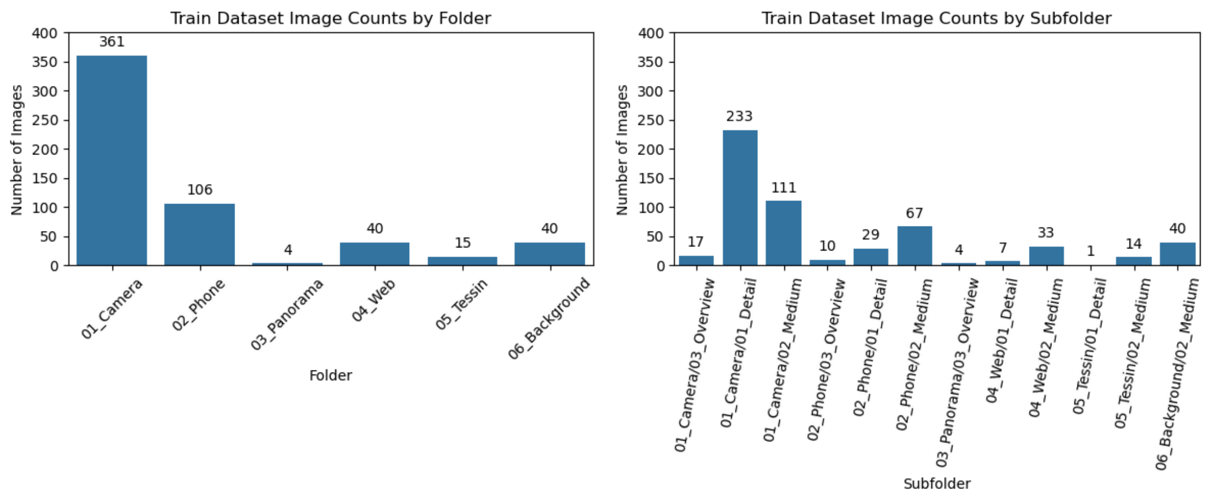


Figure 29: Count of images in the training folder

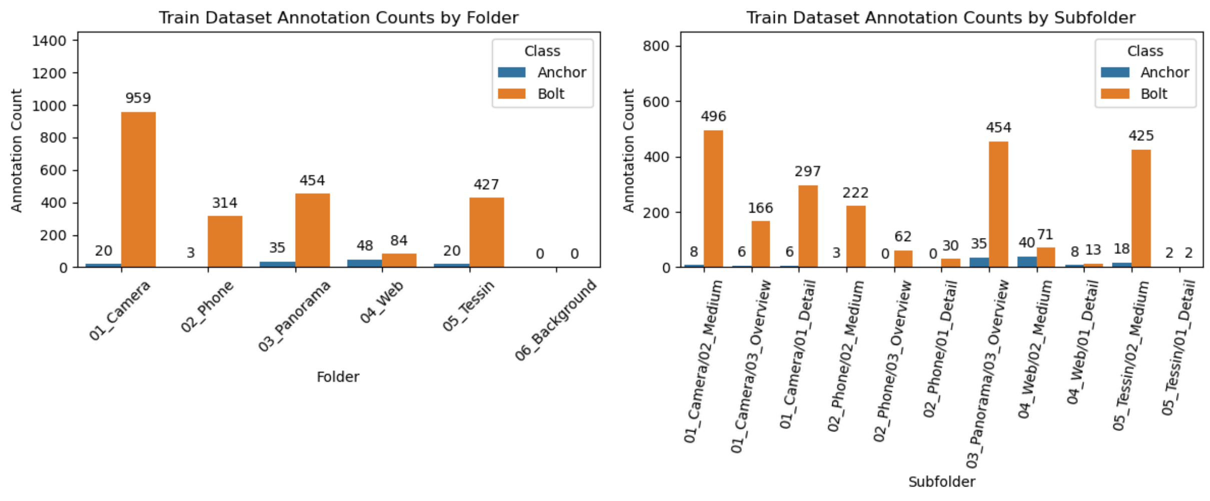


Figure 30: Count of images in the training folder

II 5.1.4.2 Test Folder

For the test dataset the left sector of the climbing sector “Chämiloch” in Schwyz is used. The following structure is used for the test folder:

- Camera
  - close ups
  - medium
  - overview
- Phone
  - close ups
  - medium
  - overview
- Panorama
  - overview

The test folder contains 266 images and 1757 bolts and 143 anchors as shown in Figure 31 and Figure 32.

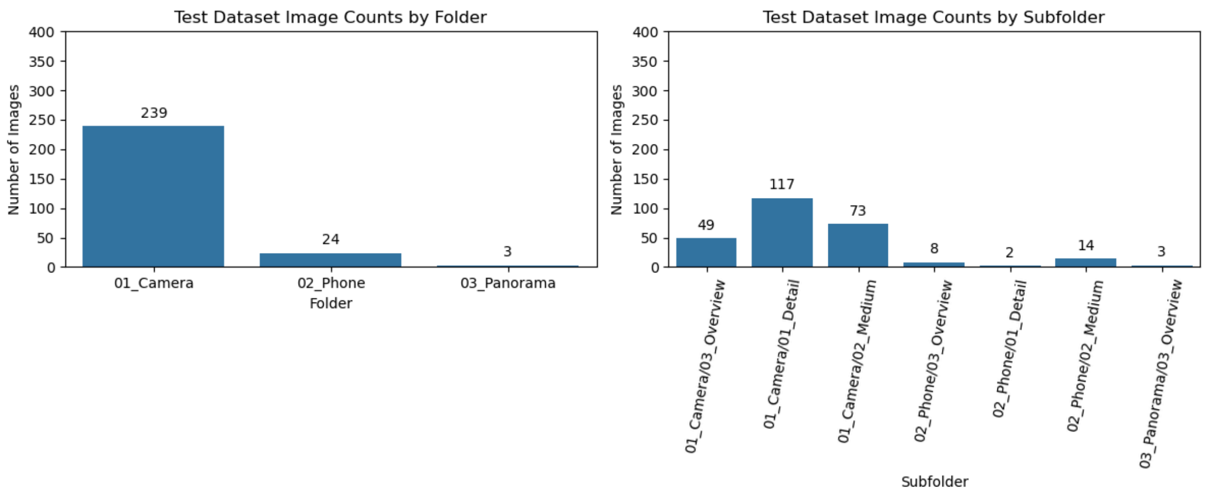


Figure 31: Count of images in the test folder

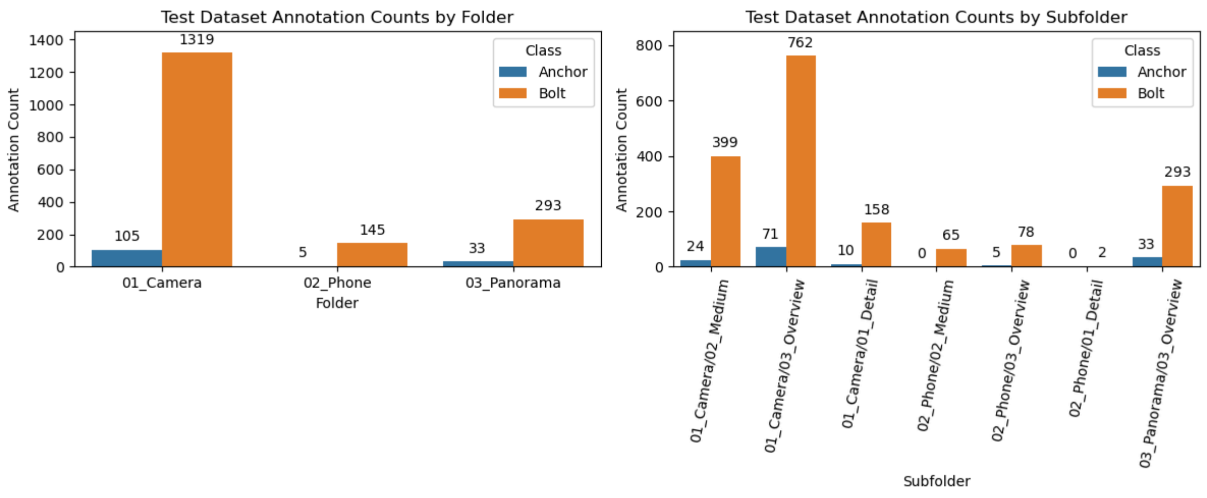


Figure 32: Count of annotations in the test folder

II 5.1.4.3 Base Dataset Analysis

The base dataset is analyzed to understand the distribution of the annotations and the images. The following sections provide insights into the dataset's characteristics.

Bolts can have various shapes and sizes, the following Figure 33 shows the variety of bolts in the dataset. Anchors are also varying in size and shape, which is already visualized in Figure 28.

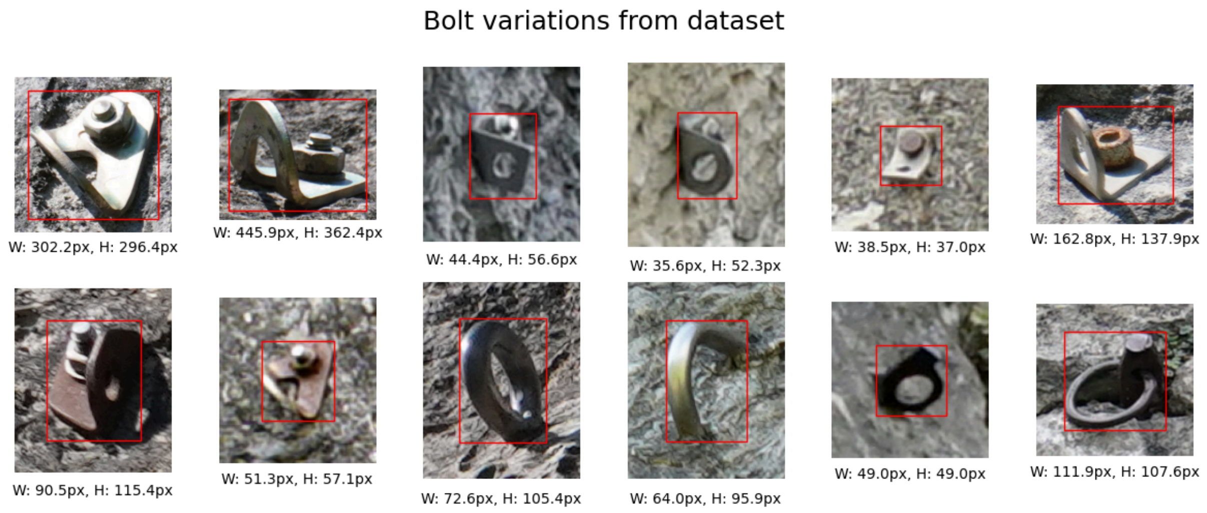


Figure 33: Bolt variations in the dataset

The datasets contain 10% background only images, which help the model to reduce false positives. A sample of those background images is shown in Figure 34.

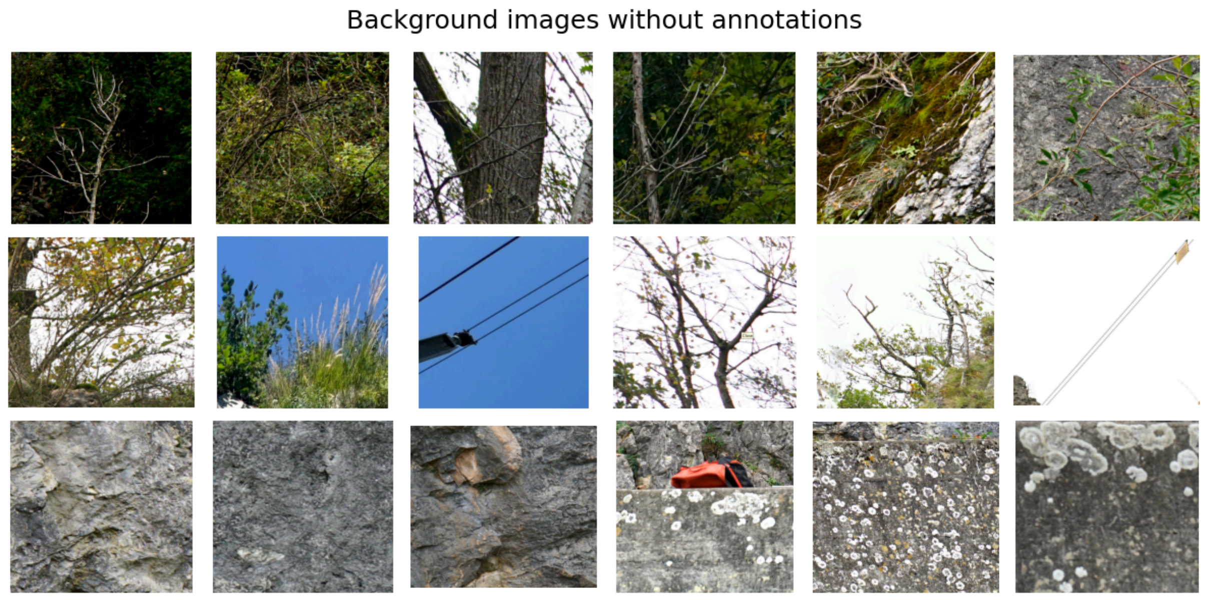


Figure 34: Background images

Plotting the heatmap of the annotations in the dataset shows that the annotations are mostly in the center of the images but also can be found scattered around the image. The heatmap is shown in Figure 35.

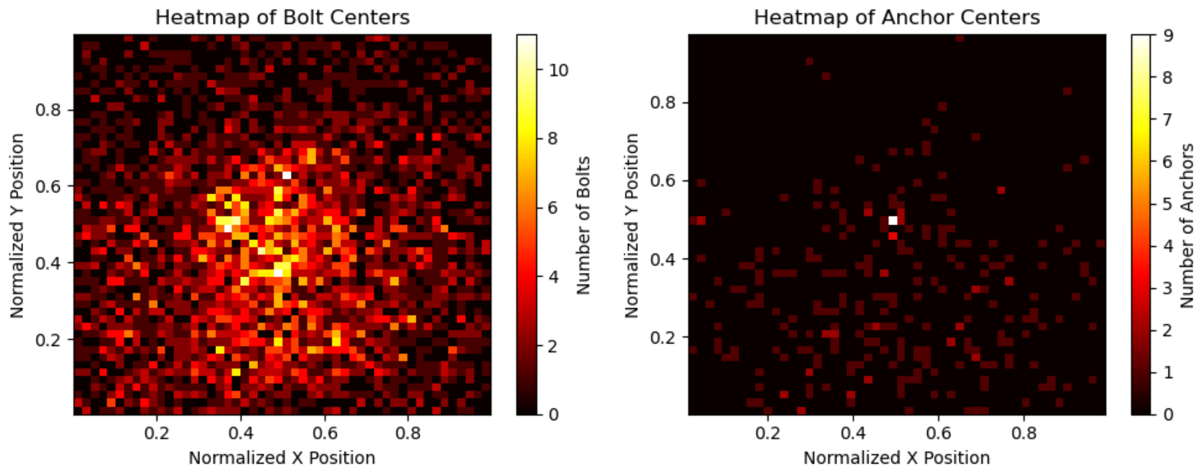


Figure 35: Heatmap of the annotations in the dataset

The annotation sizes are varying in the dataset, the following Figure 36 shows the distribution of the annotation sizes in the dataset. There are some outliers in the dataset which are very large or very small. More detailed scatter plots are in Chapter III 4.3.

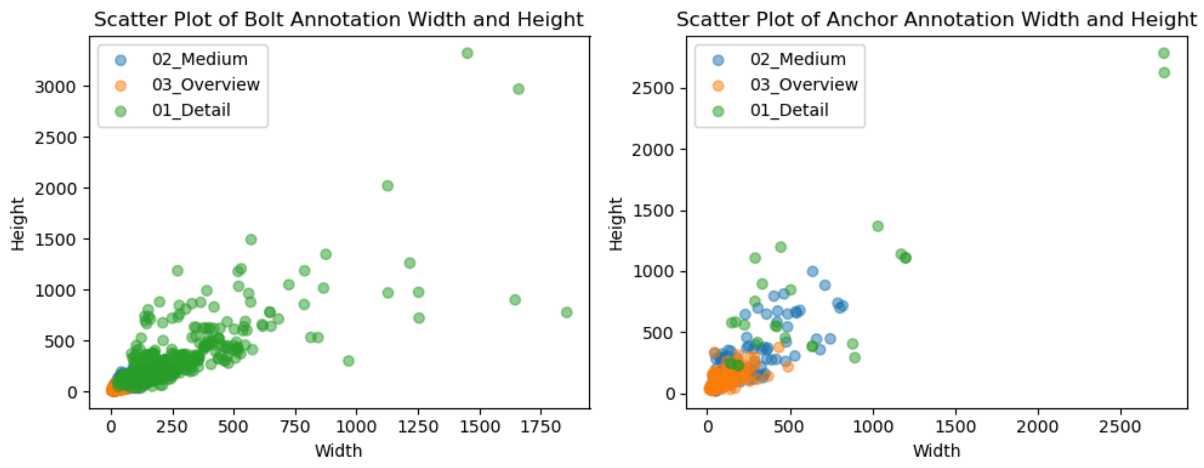


Figure 36: Annotation sizes in the dataset

The average, median as well as max and min diagonal size of the annotations is shown in Figure 37.

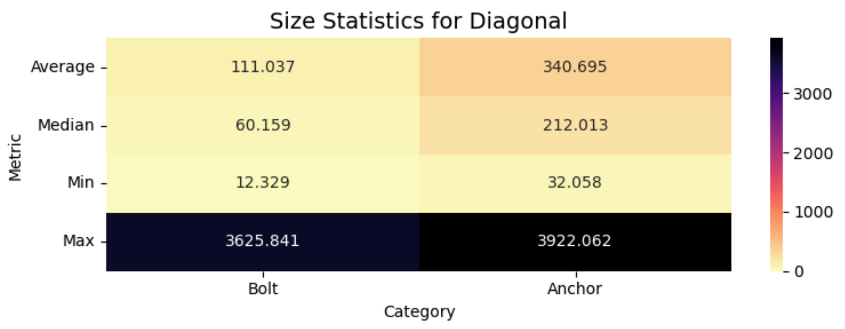


Figure 37: Annotation sizes per folder in the dataset

II 5.1.5 Creating the Training Datasets

The model repository contains the Jupyter notebook for creating training datasets at `'jupyter/training_pipeline/01_prepare_dataset.ipynb'`.

The notebook contains the following steps to create the training datasets:

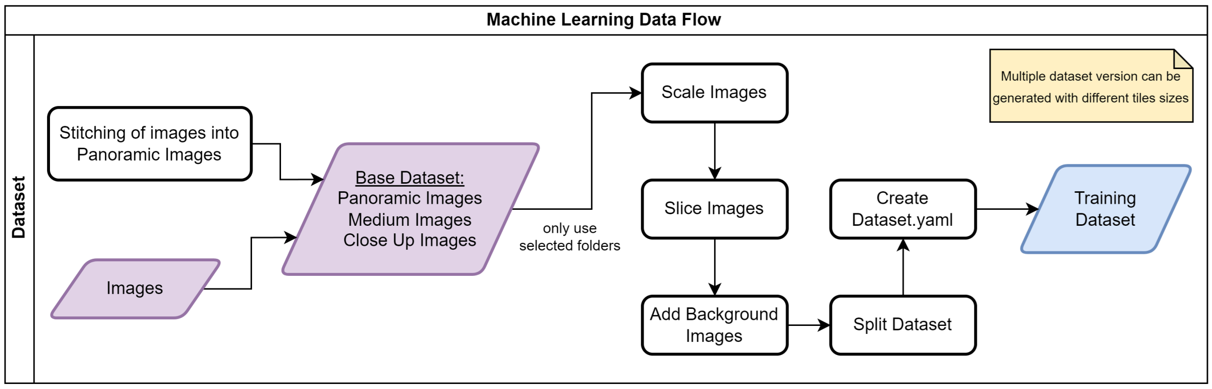

Machine Learning Data Flow
Dataset
Stitching of images into Panoramic Images
Images
Base Dataset: Panoramic Images Medium Images Close Up Images
only use selected folders
Scale Images
Slice Images
Add Background Images
Split Dataset
Create Dataset.yaml
Training Dataset
Multiple dataset version can be generated with different tiles sizes

Figure 38: High-level data flow for creating training datasets

1. Copy labeled dataset to a local directory.
2. Convert the LabelMe annotations to the YOLO format.
3. Scaling down images and annotations.
4. Slicing the images and their annotations into the tile sizes.
5. Adding 10% of background images to the dataset.
6. Splitting of images into a train and validation set.
7. Create a dataset file in the YOLO format.

In between the steps the dataset can be visualized to check if the annotations are correct and if the slicing is done correctly. Also the processed images and annotations are saved between each step to be able to continue the process at any time. The following folders will be created during the process:

- trainingdata_chämiloch_1 & custom_backgrounds contain the base dataset.
- trainingdata_yolo contains the YOLO formatted annotations and images in the base dataset structure.
- trainingdata_scaled contains the scaled down images and annotations in the base dataset structure.
- trainingdata_sliced contains the sliced images and annotations all in one folder.
- trainingdata_split contains the train and validation set, split from the sliced images.
- visualized_slices contains the visualized slices of the images and annotations.

II 5.1.5.1 Scaling of Images and Annotations

Scaling down images and annotations ensures uniformity in annotation sizes, which is crucial for the model to learn effectively. Since prediction images will consistently be taken from a similar distance from the wall, maintaining an average width and height for annotations improves model accuracy and generalization.

The scaling function is implemented in the notebook and follows these rules:

- only done for configured folders (01_Detail & 02_Medium).
- the scaling factor is calculated by the average width of the annotations.
- if the largest bolt annotation is smaller than a configured threshold, the image is not scaled.
- images from the web are always scaled.

Another approach is to scale the annotations to a fixed size, but this is not done because the model should be trained on a variety of bolt sizes.

II 5.1.5.2 Slicing the Images

Slicing is done because most of the base dataset images are too large to be directly processed by the model. Images like the panorama images contain around 100 annotations, these will then be split into more smaller sliced images of the panorama.

The main problem with slicing is that annotations may be cut off when divided across tiles. The code mitigates this issue by ensuring:
- Each tile includes annotations fully contained within its bounds.
- Overlapping or partially included annotations are handled through dynamic adjustments, such as centering tiles around remaining annotations after the initial slicing.

The code follows these steps to balance slicing efficiency and annotation preservation:

1. **Initial Pass**: The image is divided into tiles, and annotations that clearly fit within each tile are processed and saved.
2. **Handling Edge Cases**: A second pass addresses annotations near tile edges or spanning multiple tiles. Additional tiles are dynamically centered around these annotations to ensure inclusion.
3. **Refinement**: A final iterative loop processes any remaining annotations, adjusting for overlaps and gaps. Overlapping tile generation provides additional robustness against annotation loss.

Despite these efforts, some annotations may still be lost if they do not fit into any tile. Overlapping tiles and iterative refinement help minimize these losses, but they cannot be completely avoided. The code will log any images where annotations are lost.

The below Figure 39 shows the slices of a image with annotations, the annotations are colored in red and the slices are shown in blue.

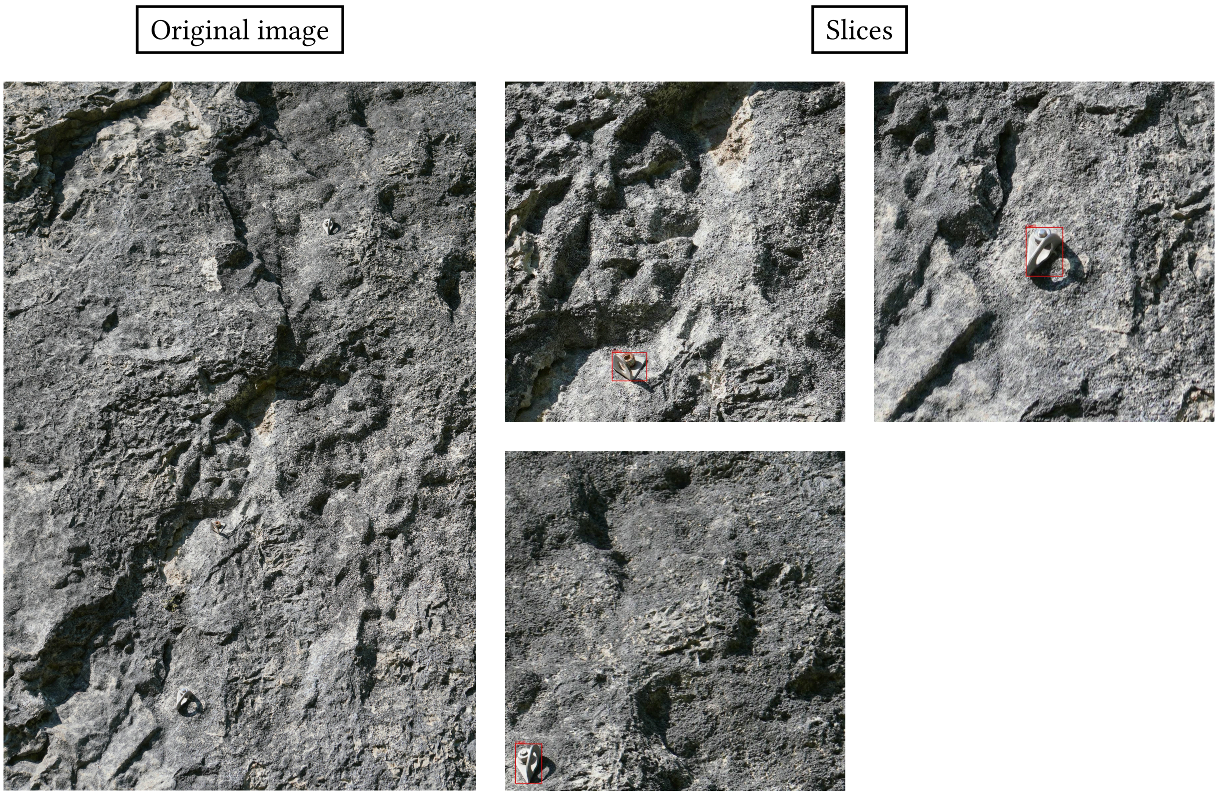


Figure 39: Camera_MidRange_DSC03048.JPG and its slices (1024px)

II 5.1.5.3 Adding Background Images

Adding background images is crucial to ensure the model does not overfit to only annotated objects and can distinguish between relevant and irrelevant parts of the images. The following steps are taken to add background images:

A configured percentage (10%) of background images is added to the dataset. These images are randomly selected from the background folder and included in the training and validation sets. If the background folder does not contain enough images to meet this requirement, random slices from the base dataset that do not contain annotations are used to fill the gap. The background images are resized to match the dimensions of the annotated slices.

The below Figure 40 shows a background slice of the image from Figure 39.

Figure 40: Sliced image with annotations

II 5.1.6 Training Dataset Versions

There are multiple versions of the training dataset depending on the training run, tile size and the training and validation split. In order to keep track of what is contained in a training dataset the dataset.yaml file contains metadata about the dataset, see Code-Fragment 4.

The following sections describe these training dataset versions.
- **Version 11**: used for YOLOv11 vs Faster R-CNN comparison.
- **Version 17 (512px, 800px, 1024px)**: used for the best tile size comparison.
- **Version 18**: used for the full dataset training.

II 5.1.6.1 Version 11

This version is created without comments in the dataset.yaml file. The dataset includes a smaller number of images and annotations than the datasets from version 17, because it used a earlier version of the base dataset. It used the folders 01_Camera, 02_Phone, 03_Panorama and 04_Web from the left sector for training and validation. The parameters like scaling and slicing are the same as used in version 17 and 18. The dataset is used for the model Faster R-CNN vs YOLOv11 results described in Chapter II 6.1.1.1.

II 5.1.6.2 Version 17

This version is done for 3 tiling sizes: 512px, 800px, 1024px. The dataset.yaml files contain metadata about the dataset, see Code-Fragment 1, Code-Fragment 2, Code-Fragment 3. The version 17 strictly uses only the training and validation folders and not the test folders. The datasets are not exactly the same, because the tiling size impacts the number of tiles and the number of empty tiles. The amount of bolts and anchors in the datasets are also different but the amounts are similar.

Training Dataset Version 17 1024px dataset.yaml

```yaml
# Generated on: 2024-12-15 22:04:17
# -------------------- Dataset Configuration --------------------
# Dataset name: run_17_multiple_slicing_techniques_1024
# Used: Remote datasets structure:
#       01_train_right:
#           01_Camera: {'02_Medium', '01_Detail', '03_Overview'}
#           02_Phone: {'02_Medium', '01_Detail', '03_Overview'}
#           03_Panorama: {'03_Overview'}
#           04_Web: {'02_Medium', '01_Detail'}
#           05_Tessin: {'02_Medium', '01_Detail'}
# Subfolders for images and labels:
#     ['images', 'labelme']
# Tile slicing parameters:
#     Tile size: (1024, 1024)
# Scale factors for folders:
#     01_Detail: 0.22
#     02_Medium: 0.77
# Maximum annotation sizes:
#     bolt: 150
#     anchor: 400
# Background tiles:
#     0.1 configured and Total number of empty slices created: 155 out of 1559 total files (9.94%)

# --------------------- Dataset Analysis ---------------------
# Train folder: 1369 images, 1782 bolts, 101 anchors, 132 empty annotation files
# Val folder: 345 images, 437 bolts, 25 anchors, 34 empty annotation files

# Train Bolts size [width, height]:
#     AVG  size: [41.19896055 49.06378166]
#     MEAN size: [33.45682148 39.16048305]
#     MIN  size: [3.05555556 6.36614173]
#     MAX  size: [365.2        730.96774194]

# Train Anchors size [width, height]:
#     AVG  size: [185.78045307 214.56885372]
#     MEAN size: [147.74774775 174.32188065]
#     MIN  size: [20.62937063 13.0625    ]
#     MAX  size: [631.6504065  766.88888889]

# Val Bolts size [width, height]:
#     AVG  size: [41.53886332 49.16761749]
#     MEAN size: [34.14782609 42.        ]
#     MIN  size: [5.11182109 8.27504554]
#     MAX  size: [276.54929577 328.50847458]

# Val Anchors size [width, height]:
#     AVG  size: [144.36693066 188.66213701]
#     MEAN size: [ 91.98378378 138.7312187 ]
#     MIN  size: [36.0669145  33.67799114]
#     MAX  size: [608.69343066 612.47786785]
names:
- Bolt
- Anchor
nc: 2
train: ./train
val: ./val
```

Code-Fragment 1: Training Dataset Version 17 1024 dataset.yaml

The dataset.yaml file for the 800px tiling size is shown in the following Code-Fragment 2.

Training Dataset Version 17 800px dataset.yaml

```yaml
# Generated on: 2024-12-15 17:39:49
# -------------------- Dataset Configuration --------------------
# Dataset name: run_17_multiple_slicing_techniques_800
# Used: Remote datasets structure:
#       01_train_right:
#           01_Camera: {'02_Medium', '01_Detail', '03_Overview'}
#           02_Phone: {'02_Medium', '01_Detail', '03_Overview'}
#           03_Panorama: {'03_Overview'}
#           04_Web: {'02_Medium', '01_Detail'}
#           05_Tessin: {'02_Medium', '01_Detail'}
# Subfolders for images and labels:
#     ['images', 'labelme']
# Tile slicing parameters:
#     Tile size: (800, 800)
# Scale factors for folders:
#     01_Detail: 0.22
#     02_Medium: 0.77
# Maximum annotation sizes:
#     bolt: 150
#     anchor: 400
# Background tiles:
#     0.1 configured and Total number of empty slices created: 171 out of 1712 total files (9.99%)

# ---------------------- Dataset Analysis ----------------------
# Train folder: 1504 images, 1758 bolts, 99 anchors, 145 empty annotation files
# Val folder: 379 images, 455 bolts, 27 anchors, 37 empty annotation files

# Train Bolts size [width, height]:
#     AVG  size: [41.00247398 48.28251721]
#     MEAN size: [33.51706529 39.70536502]
#     MIN  size: [3.05555556 7.        ]
#     MAX  size: [319.35483871 730.96774194]

# Train Anchors size [width, height]:
#     AVG  size: [177.70078711 198.28561082]
#     MEAN size: [128.5      162.004662]
#     MIN  size: [20.62937063 13.0625    ]
#     MAX  size: [618.56118331 766.88888889]

# Val Bolts size [width, height]:
#     AVG  size: [43.08266075 52.94954362]
#     MEAN size: [35.         42.34234234]
#     MIN  size: [5.40802676 6.36614173]
#     MAX  size: [365.2 653.4]

# Val Anchors size [width, height]:
#     AVG  size: [177.06004085 250.28637703]
#     MEAN size: [148.85057471 194.9044586 ]
#     MIN  size: [21.77951002 21.26503341]
#     MAX  size: [631.6504065  625.54545455]
names:
- Bolt
- Anchor
nc: 2
train: ./train
val: ./va
```

Code-Fragment 2: Training Dataset Version 17 800 dataset.yaml

The dataset.yaml file for the 512px tiling size is shown in the following Code-Fragment 3.

Training Dataset Version 17 512px dataset.yaml

```
# Generated on: 2024-12-15 19:35:30
# -------------------- Dataset Configuration --------------------
# Dataset name: run_17_multiple_slicing_techniques_512
# Used: Remote datasets structure:
#       01_train_right:
#           01_Camera: {'02_Medium', '01_Detail', '03_Overview'}
#           02_Phone: {'02_Medium', '01_Detail', '03_Overview'}
#           03_Panorama: {'03_Overview'}
#           04_Web: {'02_Medium', '01_Detail'}
#           05_Tessin: {'02_Medium', '01_Detail'}
# Subfolders for images and labels:
#     ['images', 'labelme']
# Tile slicing parameters:
#     Tile size: (512, 512)
# Scale factors for folders:
#     01_Detail: 0.22
#     02_Medium: 0.77
# Maximum annotation sizes:
#     bolt: 150
#     anchor: 400
# Background tiles:
#     0.1 configured and Total number of empty slices created: 190 out of 1903 total files (9.98%)

# ---------------------- Dataset Analysis ----------------------
# Train folder: 1672 images, 1757 bolts, 88 anchors, 160 empty annotation files
# Val folder: 421 images, 441 bolts, 23 anchors, 41 empty annotation files

# Train Bolts size [width, height]:
#     AVG  size: [40.14346196 47.0793831 ]
#     MEAN size: [33.06470588 38.66666667]
#     MIN  size: [3.05555556 6.36614173]
#     MAX  size: [276.03773585 328.50847458]

# Train Anchors size [width, height]:
#     AVG  size: [144.83775706 166.00214077]
#     MEAN size: [121.19193143 129.21124031]
#     MIN  size: [20.62937063 13.06498195]
#     MAX  size: [490.40697674 505.        ]

# Val Bolts size [width, height]:
#     AVG  size: [41.68513626 50.35439689]
#     MEAN size: [34.21052632 41.23671498]
#     MIN  size: [4.56273764 7.12401575]
#     MAX  size: [276.54929577 444.58333333]

# Val Anchors size [width, height]:
#     AVG  size: [149.08785752 192.04770949]
#     MEAN size: [ 98.7480916 194.9044586]
#     MIN  size: [27.33643123 13.0625    ]
#     MAX  size: [374.18181818 448.95797872]
names:
- Bolt
- Anchor
nc: 2
train: ./train
val: ./val
```

Code-Fragment 3: Training Dataset Version 17 512 dataset.yaml

II 5.1.6.3 Version 18

This is the training dataset that uses all of the base dataset folders for training and validation. The dataset.yaml file contains metadata about the dataset, see Code-Fragment 4. This dataset contains much more images and annotations than the datasets from version 17. The dataset is used for the model results described in Chapter II 6.1.1.4.

Training Dataset Version 18 dataset.yaml

```yaml
# Generated on: 2024-12-17 13:22:32
# -------------------- Dataset Configuration --------------------
# Dataset name: run_18_full_dataset_1024
# Used: Remote datasets structure:
#       01_train_right:
#           01_Camera: {'02_Medium', '03_Overview', '01_Detail'}
#           02_Phone: {'02_Medium', '03_Overview', '01_Detail'}
#           03_Panorama: {'03_Overview'}
#           04_Web: {'02_Medium', '01_Detail'}
#           05_Tessin: {'02_Medium', '01_Detail'}
#       02_validation_left:
#           01_Camera: {'02_Medium', '03_Overview', '01_Detail'}
#           02_Phone: {'02_Medium', '03_Overview', '01_Detail'}
#           03_Panorama: {'03_Overview'}
# Subfolders for images and labels:
#     ['images', 'labelme']
# Tile slicing parameters:
#     Tile size: (1024, 1024)
# Scale factors for folders:
#     01_Detail: 0.22
#     02_Medium: 0.77
# Maximum annotation sizes:
#     bolt: 150
#     anchor: 400
# Background tiles:
#     0.1 configured and Total number of empty slices created: 294 out of 2948 total files (9.97%)

# --------------------- Dataset Analysis ---------------------
# Train folder: 2592 images, 3132 bolts, 212 anchors, 267 empty annotation files
# Val folder: 650 images, 832 bolts, 56 anchors, 67 empty annotation files

# Train Bolts size [width, height]:
#     AVG  size: [42.0010526  46.04288229]
#     MEAN size: [34.17910638 37.18305061]
#     MIN  size: [3.05555556 6.36614173]
#     MAX  size: [408.21052632 730.96774194]

# Train Anchors size [width, height]:
#     AVG  size: [147.21085917 157.74233569]
#     MEAN size: [114.70985155 116.97310415]
#     MIN  size: [11.85185185 13.06498195]
#     MAX  size: [631.6504065  766.88888889]

# Val Bolts size [width, height]:
#     AVG  size: [40.44059013 43.01612242]
#     MEAN size: [33.9        35.15914475]
#     MIN  size: [4.56273764 7.86796785]
#     MAX  size: [276.03773585 444.58333333]

# Val Anchors size [width, height]:
#     AVG  size: [147.53581115 167.69863986]
#     MEAN size: [121.74603175 123.3516401 ]
#     MIN  size: [22.27564103 13.0625    ]
#     MAX  size: [548.96470588 680.31764706]
names:
- Bolt
- Anchor
nc: 2
train: ./train
val: ./val
```

Code-Fragment 4: Training Dataset Version 18 dataset.yaml

II 5.1.7 Training & Testing Panoramic Dataset

The panoramic images represent the images which will be uploaded to the BoltFinder web application. These panoramic images are stitched together using the Microsoft Image Composite Editor see Chapter II 3.4.3. These images take up more space than usual ones because of their size they span from 23,476px to 49,301px in width and from 124MB to 827MB in file size. In Figure 41 the left sector is shown from three angles, this is the set that is used to test the models which are trained on the data from the right sector only.

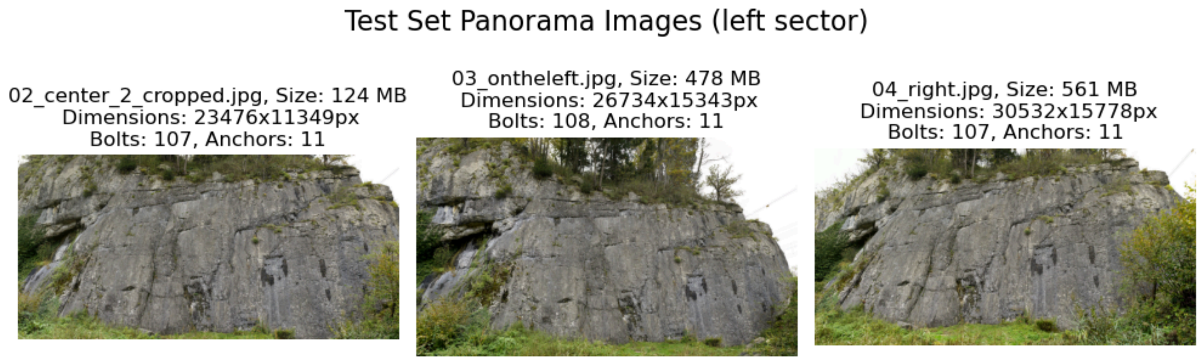

Test Set Panorama Images (left sector)
02_center_2_cropped.jpg, Size: 124 MB
Dimensions: 23476x11349px
Bolts: 107, Anchors: 11
03_ontheleft.jpg, Size: 478 MB
Dimensions: 26734x15343px
Bolts: 108, Anchors: 11
04_right.jpg, Size: 561 MB
Dimensions: 30532x15778px
Bolts: 107, Anchors: 11

Figure 41: Panoramic images of the climbing sector Chämiloch

The Figure 42 shows the four walls from the right sector, this is part of the set that is used to train the models.

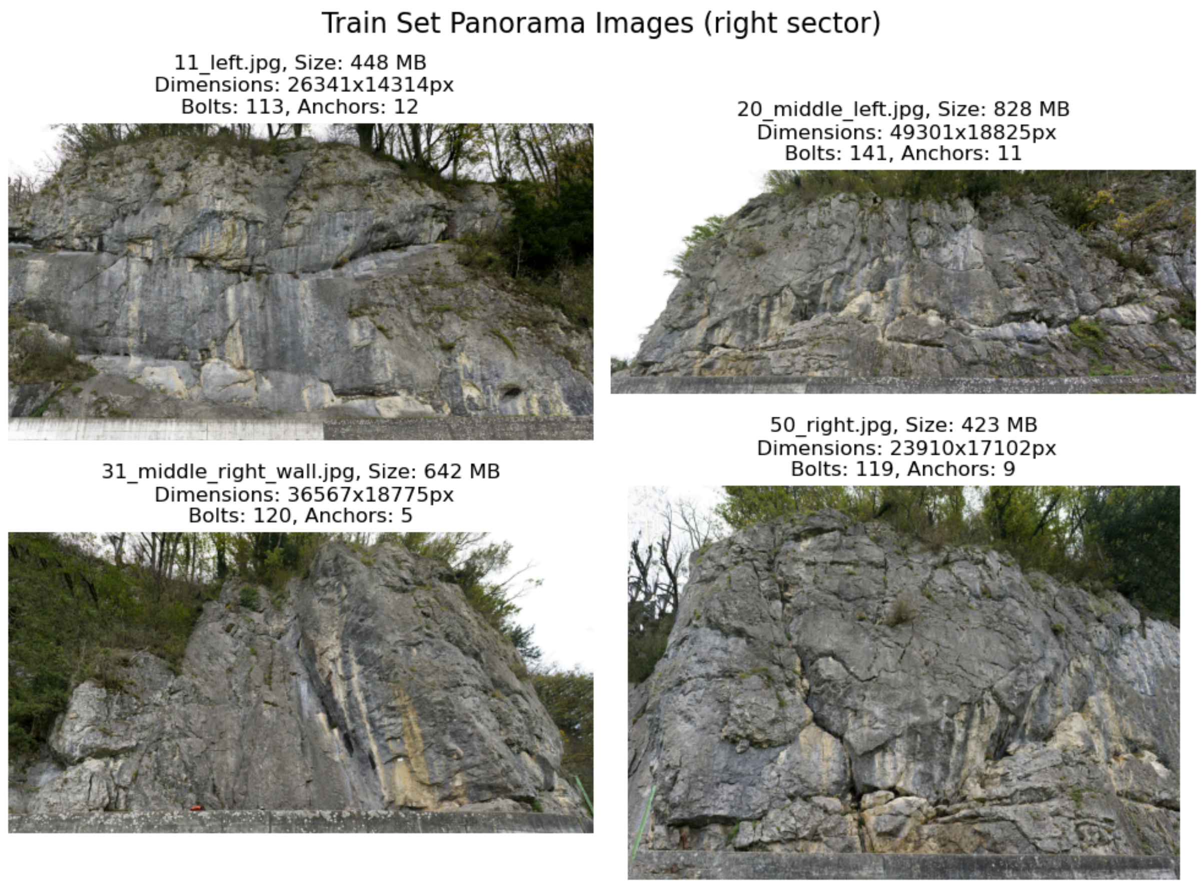

Train Set Panorama Images (right sector)
11_left.jpg, Size: 448 MB
Dimensions: 26341x14314px
Bolts: 113, Anchors: 12
20_middle_left.jpg, Size: 828 MB
Dimensions: 49301x18825px
Bolts: 141, Anchors: 11
31_middle_right_wall.jpg, Size: 642 MB
Dimensions: 36567x18775px
Bolts: 120, Anchors: 5
50_right.jpg, Size: 423 MB
Dimensions: 23910x17102px
Bolts: 119, Anchors: 9

Figure 42: Panoramic images of the climbing sector Chämiloch

Both sets are used for training and testing the combined model (v18) which is trained on all available data.

II 5.2 ML Object Detection

A big part of the project is the object detection of the bolts. The object detection is implemented using the YOLOv11 and Faster R-CNN models. The models are trained on the dataset and the results are compared in Chapter II 6.1.1. This chapter presents the workflow of training new models.

II 5.2.1 Training Workflow

The training workflow consists of multiple Jupyter notebooks that are used to preprocess the data, train the model and save the model.

- `01_prepare_dataset.ipynb`
- `03_train_rcnn.ipynb`
- `03_train_yolo.ipynb`

In the notebooks parameters like epochs, learning rate or run name can be set before running them manually. The workflow is implemented using the following steps shown in Figure 43 steps:

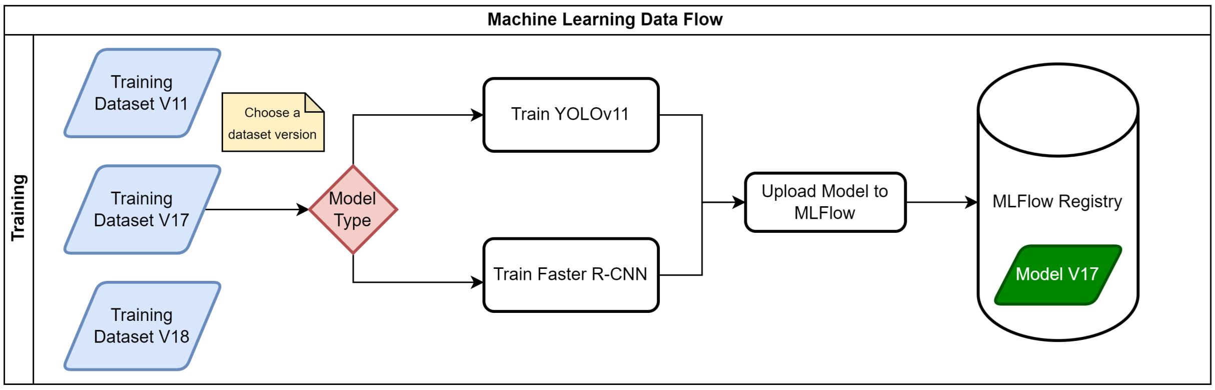


Figure 43: High-level data for object detection training

The creation of training datasets is described in detail in Chapter II 5.1.5

- Choose the model to train

- Model Training YOLO
  - Create a new MLFlow run
  - Train the model using the yolox.pt model
  - Log metrics each epoch to MLFlow

- Model Training Faster R-CNN
  - Create a new MLFlow run
  - Train the model using the yolox.pt model
  - Log metrics each epoch to MLFlow

- Model Saving
  - Save the Model locally
  - Upload the new model to the MLFlow registry

II 5.2.2 Training Models

The models used for the bolt recognition are the Faster R-CNN and YOLOv11 models. The models are trained on the dataset and the results are compared in Chapter II 6.1.1.

II 5.2.2.1 YOLOv11 Model

YOLOv11 is a pretrained model provided by the Ultralytics library. The models are trained on the various training datasets and the results are compared in Chapter II 6.1.1. The YOLOv11 is the latest version of the YOLO model and is optimized for efficient object detection. As shown in Figure 44.

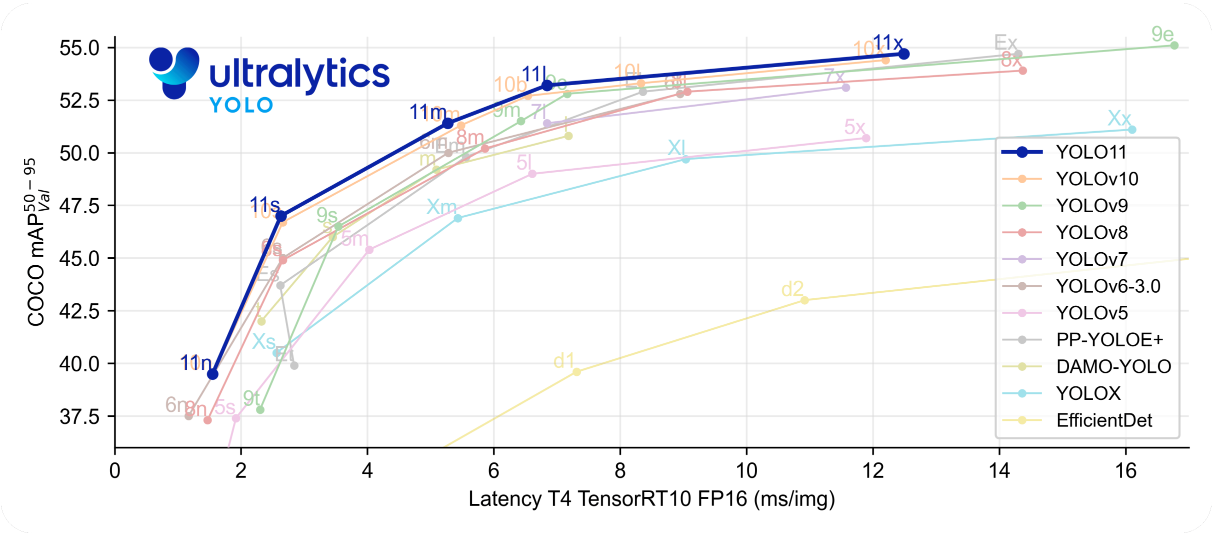


Figure 44: YOLOv11 map50-95 Performance Comparison [27]

The YOLOv11 model is available in different sizes, ranging from small to extra-large. The model sizes are compared in the following Table 18. The YOLOv11x is used because it has the highest mAP value and is the largest model.

Model	Size (pixels)	mAPval 50-95	Speed (CPU ONNX, ms)	Speed (T4 TensorRT10, ms)	Params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

Table 18: YOLOv11 Model Sizes [27]

II 5.2.2.1.1 Model Layers

The model layers are listed in the following Code-Fragment 5 and are generated by the YOLOv11x (extra-large) model when training the model.

YOLOv11x layers

```
         from  n    params  module                                       arguments
           -1  1      2784  ultralytics.nn.modules.conv.Conv             [3, 96, 3, 2]
           -1  1    166272  ultralytics.nn.modules.conv.Conv             [96, 192, 3, 2]
           -1  2    389760  ultralytics.nn.modules.block.C3k2            [192, 384, 2, True, 0.25]
           -1  1   1327872  ultralytics.nn.modules.conv.Conv             [384, 384, 3, 2]
           -1  2   1553664  ultralytics.nn.modules.block.C3k2            [384, 768, 2, True, 0.25]
           -1  1   5309952  ultralytics.nn.modules.conv.Conv             [768, 768, 3, 2]
           -1  2   5022720  ultralytics.nn.modules.block.C3k2            [768, 768, 2, True]
           -1  1   5309952  ultralytics.nn.modules.conv.Conv             [768, 768, 3, 2]
           -1  2   5022720  ultralytics.nn.modules.block.C3k2            [768, 768, 2, True]
           -1  1   1476864  ultralytics.nn.modules.block.SPPF            [768, 768, 5]
           -1  2   3264768  ultralytics.nn.modules.block.C2PSA           [768, 768, 2]
           -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
      [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat           [1]
           -1  2   5612544  ultralytics.nn.modules.block.C3k2            [1536, 768, 2, True]
           -1  1         0  torch.nn.modules.upsampling.Upsample         [None, 2, 'nearest']
      [-1, 4]  1         0  ultralytics.nn.modules.conv.Concat           [1]
           -1  2   1700352  ultralytics.nn.modules.block.C3k2            [1536, 384, 2, True]
           -1  1   1327872  ultralytics.nn.modules.conv.Conv             [384, 384, 3, 2]
     [-1, 13]  1         0  ultralytics.nn.modules.conv.Concat           [1]
           -1  2   5317632  ultralytics.nn.modules.block.C3k2            [1152, 768, 2, True]
           -1  1   5309952  ultralytics.nn.modules.conv.Conv             [768, 768, 3, 2]
     [-1, 10]  1         0  ultralytics.nn.modules.conv.Concat           [1]
           -1  2   5612544  ultralytics.nn.modules.block.C3k2            [1536, 768, 2, True]
 [16, 19, 22]  1   3147862  ultralytics.nn.modules.head.Detect           [2, [384, 768, 768]]
YOLO11x summary: 631 layers, 56,876,086 parameters, 56,876,070 gradients
```

Code-Fragment 5: YOLOv11 X model layers

II 5.2.2.1.2 Training Procedure

To train the YOLOv11 model, the Ultralytics library is used. The training process is shown in the following code snippet Table 19. The model is trained with augmentation parameters to improve the model's performance. Other parameters like epochs, batch size, and image size are also set during training.

YOLOv11 training with augmentation parameters

```python
model = YOLO("yolo11x.pt")
results = model.train(
  data=dataset,
  epochs=epochs,
  batch=batchsize,
  imgsz=imgsz,
  name=runStamp,
  degrees=45,
  shear=5
)
```

Code-Fragment 6: YOLOv11 X model layers

II 5.2.2.1.3 Yolo Augmentation Techniques

The Ultralytics library provides built in augmentation techniques for the dataset [27]. The augmentation techniques are listed in the following Table 19 and are used for training the model. Each technique is applied to an image and label per epoch.

Argument	Type	Default	Range	Description
hsv_h	float	0.015	0.0 - 1.0	Adjusts the hue of the image by a fraction of the color wheel, introducing color variability. Helps the model generalize across different lighting conditions.
hsv_s	float	0.7	0.0 - 1.0	Alters the saturation of the image by a fraction, affecting the intensity of colors. Useful for simulating different environmental conditions.
hsv_v	float	0.4	0.0 - 1.0	Modifies the value (brightness) of the image by a fraction, helping the model to perform well under various lighting conditions.
degrees	float	0.0	−180 - +180	Rotates the image randomly within the specified degree range, improving the model's ability to recognize objects at various orientations.
translate	float	0.1	0.0 - 1.0	Translates the image horizontally and vertically by a fraction of the image size, aiding in learning to detect partially visible objects.
scale	float	0.5	>=0.0	Scales the image by a gain factor, simulating objects at different distances from the camera.
shear	float	0.0	−180 - +180	Shears the image by a specified degree, mimicking the effect of objects being viewed from different angles.
perspective	float	0.0	0.0 - 0.001	Applies a random perspective transformation to the image, enhancing the model's ability to understand objects in 3D space.
flipud	float	0.0	0.0 - 1.0	Flips the image upside down with the specified probability, increasing the data variability without affecting the object's characteristics.
fliplr	float	0.5	0.0 - 1.0	Flips the image left to right with the specified probability, useful for learning symmetrical objects and increasing dataset diversity.
bgr	float	0.0	0.0 - 1.0	Flips the image channels from RGB to BGR with the specified probability, useful for increasing robustness to incorrect channel ordering.
mosaic	float	1.0	0.0 - 1.0	Combines four training images into one, simulating different scene compositions and object interactions. Highly effective for complex scene understanding.
mixup	float	0.0	0.0 - 1.0	Blends two images and their labels, creating a composite image. Enhances the model's ability to generalize by introducing label noise and visual variability.
copy_paste	float	0.0	0.0 - 1.0	Copies and pastes objects across images, useful for increasing object instances and learning object occlusion. Requires segmentation labels.
copy_paste_mode	str	flip	-	Copy-Paste augmentation method selection among the options of ("flip", "mixup").
auto_augment	str	randaugment	-	Automatically applies a predefined augmentation policy (randaugment, autoaugment, augmix), optimizing for classification tasks by diversifying the visual features.
erasing	float	0.4	0.0 - 0.9	Randomly erases a portion of the image during classification training, encouraging the model to focus on less obvious features for recognition.
crop_fraction	float	1.0	0.1 - 1.0	Crops the classification image to a fraction of its size to emphasize central features and adapt to object scales, reducing background distractions.

Table 19: Augmentation techniques from Ultralytics for training the YOLO model [27]

II 5.2.2.2 Faster R-CNN Model

Faster R-CNN is a deep learning model tailored for object detection tasks. It combines region proposal and classification networks into a unified framework, offering efficient and accurate object detection. Below, we document the model setup and training process in detail.

II 5.2.2.2.1 Architecture and Model Initialization

The implementation uses a `ResNet-50` backbone with a Feature Pyramid Network (FPN) for multi-scale feature extraction. The `ResNet-50` backbone processes the input image and extracts hierarchical features, which are then passed through the FPN to detect objects of various sizes more effectively. The high-level architecture of Faster R-CNN is illustrated below in Figure 45:

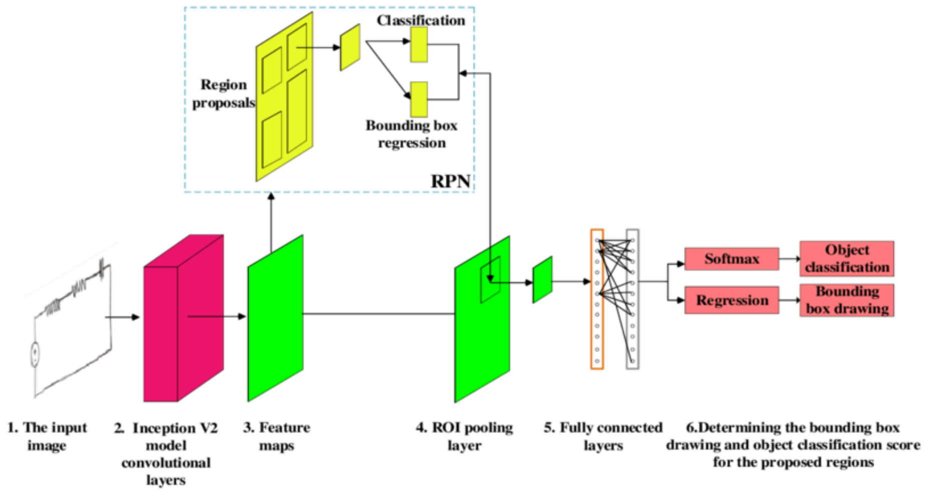


Figure 45: Faster R-CNN Architecture with Feature Extraction, Region Proposal, and Prediction Components. [28]

To customize the model for the specific dataset, the default classification and bounding box regression heads are replaced with a `FastRCNNPredictor`. This ensures compatibility with the backbone and enables the model to adapt to the defined number of classes. The initialization function used is shown here:

Faster R-CNN Model Initialization

```python
def get_model(num_classes):
    model = fasterrcnn_resnet50_fpn(weights=None)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
    return model
```

Code-Fragment 7: Faster R-CNN Model Initialization

This approach enables seamless integration of the predictor with the backbone by accessing its feature dimensions. Since pre-trained weights underperform in initial tests (with mAP values below 0.001), the model is trained from scratch (`weights=None`), which leads to better results.

II 5.2.2.2.2 Training Parameters

The training process is carefully designed to optimize the model's performance. Key parameters include:

- **Optimizer**: Stochastic Gradient Descent (SGD) with a momentum of 0.8.
- **Learning Rate**: Set to 0.005, balancing convergence speed and stability.
- **Batch Size**: A batch size of 8 ensures efficient GPU utilization.

The dataset is formatted in COCO-style annotations and preprocessed with `CocoDetection` and `ToTensor()` transformations. The complete training workflow is illustrated below in Figure 46:

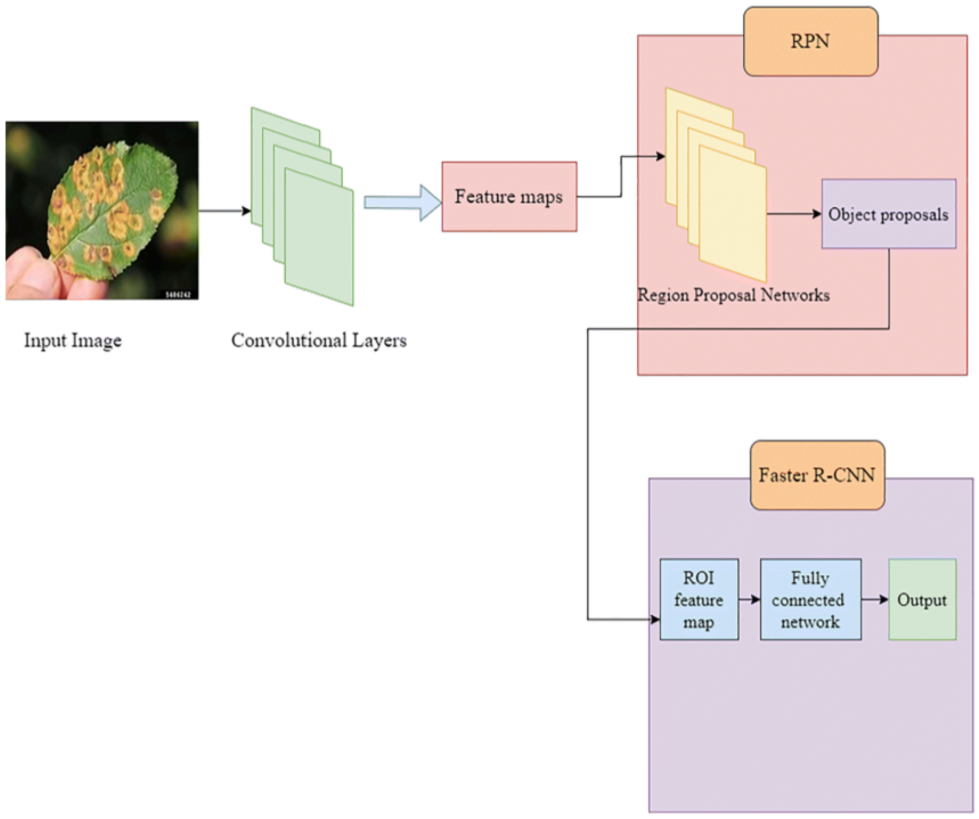


Figure 46: End-to-end Training Workflow for Faster R-CNN, showing Data Input, Loss Calculation, and Optimization Steps. [28]

II 5.2.2.2.3 Training Procedure

The model is trained iteratively, processing batches of images and annotations, calculating losses, and updating model weights. The following Code-Fragment 8 represents the main training loop:

Faster R-CNN Training Loop

```
for images, targets in train_loader:
    images = [img.to(device) for img in images]
    targets = convert_target_format(targets)
    targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

    loss_dict = model(images, targets)
    losses = sum(loss for loss in loss_dict.values())

    optimizer.zero_grad()
    losses.backward()
    optimizer.step()
```

Code-Fragment 8: Faster R-CNN Training Loop

II 5.2.2.2.4 Evaluation Metrics

To assess the model's performance during training, the COCO evaluation metrics were calculated using the `pycocotools` library. This library provides a standardized way to compute key metrics for object detection models, including Mean Average Precision (mAP) and Mean Average Recall (mAR), across various Intersection over Union (IoU) thresholds and object sizes.

The `evaluate_coco_metrics` function was used to compute these metrics, with results logged at each epoch. An example of the evaluation code is shown below in Code-Fragment 9:

COCO Evaluation Metrics Calculation

```python
from pycocotools.coco import COCO
from pycocotools.cocoeval import COCOeval

def evaluate_coco_metrics(coco_gt, coco_results):
    coco_dt = coco_gt.loadRes(coco_results)
    coco_eval = COCOeval(coco_gt, coco_dt, "bbox")
    coco_eval.evaluate()
    coco_eval.accumulate()
    coco_eval.summarize()

    metrics = {
        "AP@[IoU=0.50:0.95|area=all]": coco_eval.stats[0],
        "AP@[IoU=0.50|area=all]": coco_eval.stats[1],
        "AP@[IoU=0.75|area=all]": coco_eval.stats[2],
        "AR@[IoU=0.50:0.95|area=all]": coco_eval.stats[8],
    }
    return metrics
```

Code-Fragment 9: COCO Evaluation Code

The evaluation results, calculated after each epoch, include detailed metrics for precision and recall across different IoU thresholds, object sizes, and detection limits. Below in Code-Fragment 10 is a sample output representative of the second epoch:

Sample COCO Evaluation Output

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.050
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.208
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.009
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.007
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.079
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.049
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.110
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.231
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.291
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.200
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.346
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.289
Epoch 2/75
```

Code-Fragment 10: Sample COCO Evaluation Output

These metrics, particularly the Average Precision (AP) and Average Recall (AR) values, are critical for evaluating the model's detection quality and recall capacity. They provide valuable feedback to refine the model and improve performance across various object scales.

II 5.2.3 MLFlow Trainings and Models

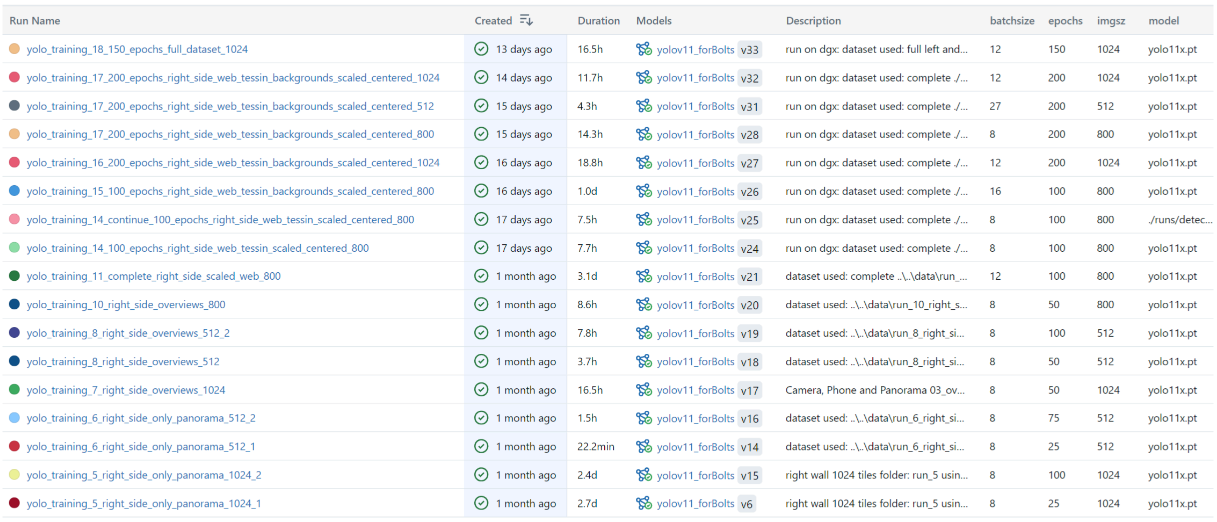
MLFlow is used to track the training of the models and to save the models in a registry. The following Code-Fragment 11 shows the functions used for logging the training to MLFlow.

Python code for logging to MLFlow

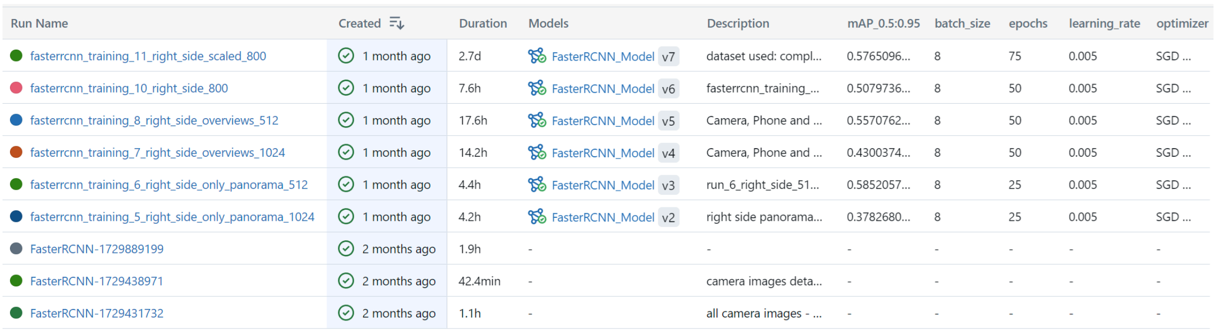
```python
mlflow.set_tracking_uri("https://mlflow-ba.infs.ch")
mlflow.set_experiment(experiment_name="YOLOv11")

with mlflow.start_run(run_name=runName, description=description) as run:
  mlflow.log_param("epochs", epochs)

  try:
    # log the metrics in try catch block to prevent stopping the training
    mlflow.log_metrics(metrics=metrics_dict, step=trainer.epoch)
  except Exception as e:
    print(f"Error logging metrics: {e}")

  mlflow.pytorch.log_model(
      model.model, "model", registered_model_name=registered_model_name
  )

  mlflow.log_artifact(model_path, "model/data")
```

Code-Fragment 11: Python code for logging to MLFlow

The following screenshots in Figure 48 show the various MLFlow Experiments (model trainings) for YOLOv11 and Faster R-CNN.



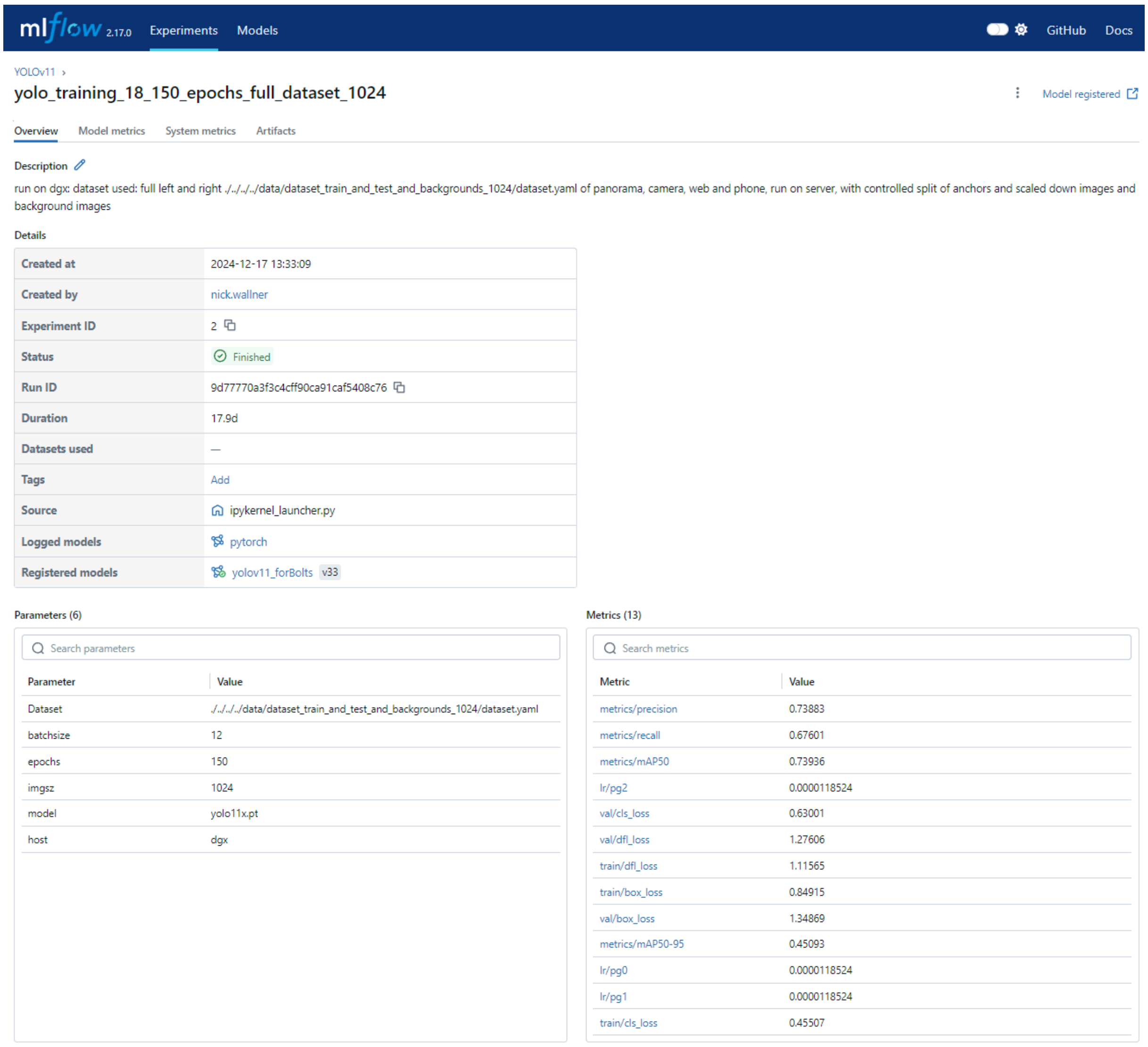
Run Name	Created	Duration	Models	Description	batchsize	epochs	imgsz	model
yolo_training_18_150_epochs_full_dataset_1024	13 days ago	16.5h	yolov11_forBolts v33	run on dgx: dataset used: full left and...	12	150	1024	yolo11x.pt
yolo_training_17_200_epochs_right_side_web_tessin_backgrounds_scaled_centered_1024	14 days ago	11.7h	yolov11_forBolts v32	run on dgx: dataset used: complete ./...	12	200	1024	yolo11x.pt
yolo_training_17_200_epochs_right_side_web_tessin_backgrounds_scaled_centered_512	15 days ago	4.3h	yolov11_forBolts v31	run on dgx: dataset used: complete ./...	27	200	512	yolo11x.pt
yolo_training_17_200_epochs_right_side_web_tessin_backgrounds_scaled_centered_800	15 days ago	14.3h	yolov11_forBolts v28	run on dgx: dataset used: complete ./...	8	200	800	yolo11x.pt
yolo_training_16_200_epochs_right_side_web_tessin_backgrounds_scaled_centered_1024	16 days ago	18.8h	yolov11_forBolts v27	run on dgx: dataset used: complete ./...	12	200	1024	yolo11x.pt
yolo_training_15_100_epochs_right_side_web_tessin_backgrounds_scaled_centered_800	16 days ago	1.0d	yolov11_forBolts v26	run on dgx: dataset used: complete ./...	16	100	800	yolo11x.pt
yolo_training_14_continue_100_epochs_right_side_web_tessin_scaled_centered_800	17 days ago	7.5h	yolov11_forBolts v25	run on dgx: dataset used: complete ./...	8	100	800	./runs/detec...
yolo_training_14_100_epochs_right_side_web_tessin_scaled_centered_800	17 days ago	7.7h	yolov11_forBolts v24	run on dgx: dataset used: complete ./...	8	100	800	yolo11x.pt
yolo_training_11_complete_right_side_scaled_web_800	1 month ago	3.1d	yolov11_forBolts v21	dataset used: complete ..\..\data\run_...	12	100	800	yolo11x.pt
yolo_training_10_right_side_overviews_800	1 month ago	8.6h	yolov11_forBolts v20	dataset used: ..\..\data\run_10_right_s...	8	50	800	yolo11x.pt
yolo_training_8_right_side_overviews_512_2	1 month ago	7.8h	yolov11_forBolts v19	dataset used: ..\..\data\run_8_right_si...	8	100	512	yolo11x.pt
yolo_training_8_right_side_overviews_512	1 month ago	3.7h	yolov11_forBolts v18	dataset used: ..\..\data\run_8_right_si...	8	50	512	yolo11x.pt
yolo_training_7_right_side_overviews_1024	1 month ago	16.5h	yolov11_forBolts v17	Camera, Phone and Panorama 03_ov...	8	50	1024	yolo11x.pt
yolo_training_6_right_side_only_panorama_512_2	1 month ago	1.5h	yolov11_forBolts v16	dataset used: ..\..\data\run_6_right_si...	8	75	512	yolo11x.pt
yolo_training_6_right_side_only_panorama_512_1	1 month ago	22.2min	yolov11_forBolts v14	dataset used: ..\..\data\run_6_right_si...	8	25	512	yolo11x.pt
yolo_training_5_right_side_only_panorama_1024_2	1 month ago	2.4d	yolov11_forBolts v15	right wall 1024 tiles folder: run_5 usin...	8	100	1024	yolo11x.pt
yolo_training_5_right_side_only_panorama_1024_1	1 month ago	2.7d	yolov11_forBolts v6	right wall 1024 tiles folder: run_5 usin...	8	25	1024	yolo11x.pt



Run Name	Created	Duration	Models	Description	mAP_0.5:0.95	batch_size	epochs	learning_rate	optimizer
fasterrcnn_training_11_right_side_scaled_800	1 month ago	2.7d	FasterRCNN_Model v7	dataset used: compl...	0.5765096...	8	75	0.005	SGD ...
fasterrcnn_training_10_right_side_800	1 month ago	7.6h	FasterRCNN_Model v6	fasterrcnn_training_...	0.5079736...	8	50	0.005	SGD ...
fasterrcnn_training_8_right_side_overviews_512	1 month ago	17.6h	FasterRCNN_Model v5	Camera, Phone and ...	0.5570762...	8	50	0.005	SGD ...
fasterrcnn_training_7_right_side_overviews_1024	1 month ago	14.2h	FasterRCNN_Model v4	Camera, Phone and ...	0.4300374...	8	50	0.005	SGD ...
fasterrcnn_training_6_right_side_only_panorama_512	1 month ago	4.4h	FasterRCNN_Model v3	run_6_right_side_51...	0.5852057...	8	25	0.005	SGD ...
fasterrcnn_training_5_right_side_only_panorama_1024	1 month ago	4.2h	FasterRCNN_Model v2	right side panorama...	0.3782680...	8	25	0.005	SGD ...
FasterRCNN-1729889199	2 months ago	1.9h	-	-	-	-	-	-	-
FasterRCNN-1729438971	2 months ago	42.4min	-	camera images deta...	-	-	-	-	-
FasterRCNN-1729431732	2 months ago	1.1h	-	all camera images - ...	-	-	-	-	-

Figure 48: MLFlow Models for YOLO & Faster R-CNN

For each training run, the training parameters, metrics and the model outputs are saved in the MLFlow tracking tool. The following screenshots show the MLFlow tracking for the YOLOv11 model from the run 18.



yolo_training_18_150_epochs_full_dataset_1024

Description

run on dgx: dataset used: full left and right ./../../../data/dataset_train_and_test_and_backgrounds_1024/dataset.yaml of panorama, camera, web and phone, run on server, with controlled split of anchors and scaled down images and background images

Details

Created at	2024-12-17 13:33:09
Created by	nick.wallner
Experiment ID	2
Status	Finished
Run ID	9d77770a3f3c4cff90ca91caf5408c76
Duration	17.9d
Datasets used	—
Tags	Add
Source	ipykernel_launcher.py
Logged models	pytorch
Registered models	yolov11_forBolts v33

Parameters (6)

Search parameters

Parameter	Value
Dataset	./../../../data/dataset_train_and_test_and_backgrounds_1024/dataset.yaml
batchsize	12
epochs	150
imgsz	1024
model	yolo11x.pt
host	dgx

Metrics (13)

Search metrics

Metric	Value
metrics/precision	0.73883
metrics/recall	0.67601
metrics/mAP50	0.73936
lr/pg2	0.0000118524
val/cls_loss	0.63001
val/dfl_loss	1.27606
train/dfl_loss	1.11565
train/box_loss	0.84915
val/box_loss	1.34869
metrics/mAP50-95	0.45093
lr/pg0	0.0000118524
lr/pg1	0.0000118524
train/cls_loss	0.45507

Figure 49: MLFlow YOLOv11 model

The Figure 50 shows the metrics like precision, recall and mAP50 for each epoch of the YOLOv11 model. Note that these metrics are based on the validation dataset and not the testing dataset.

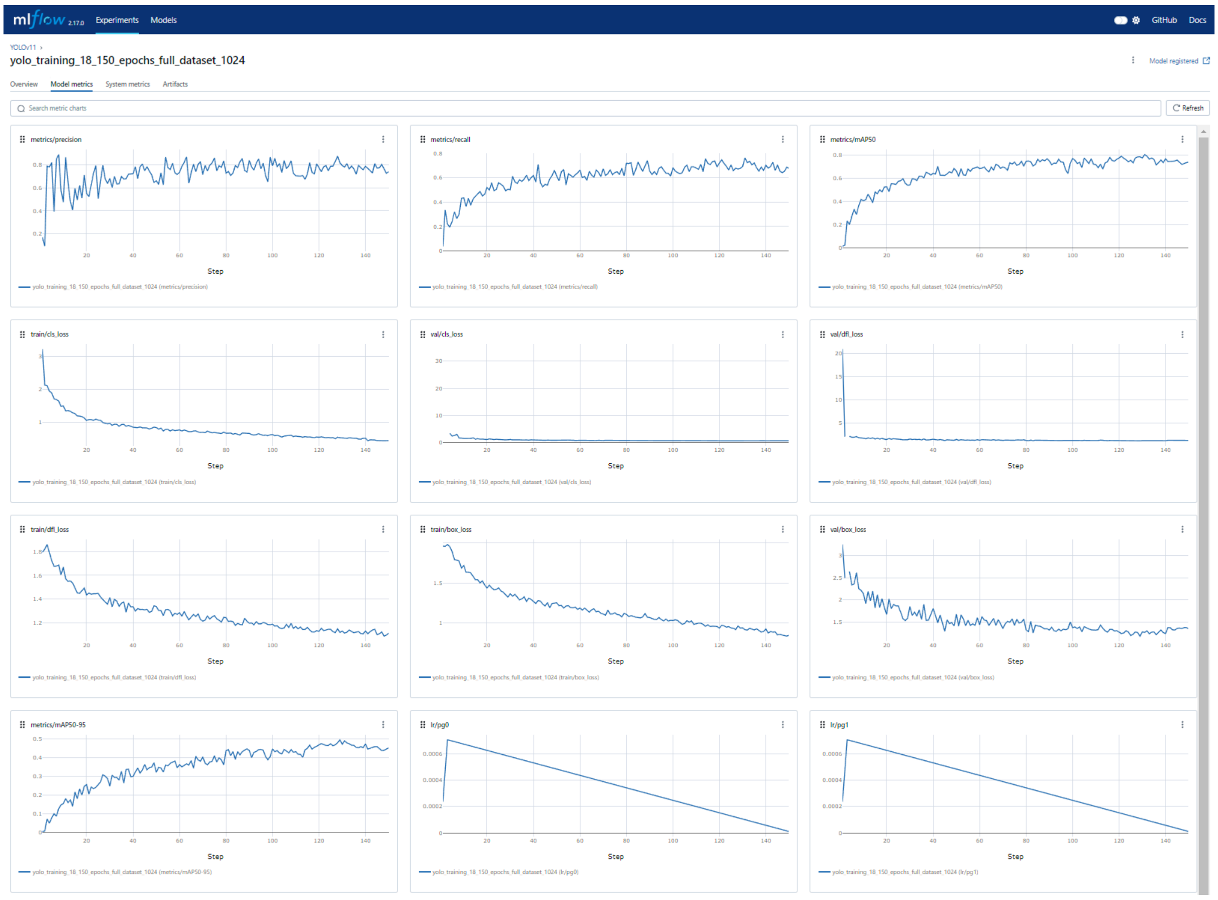
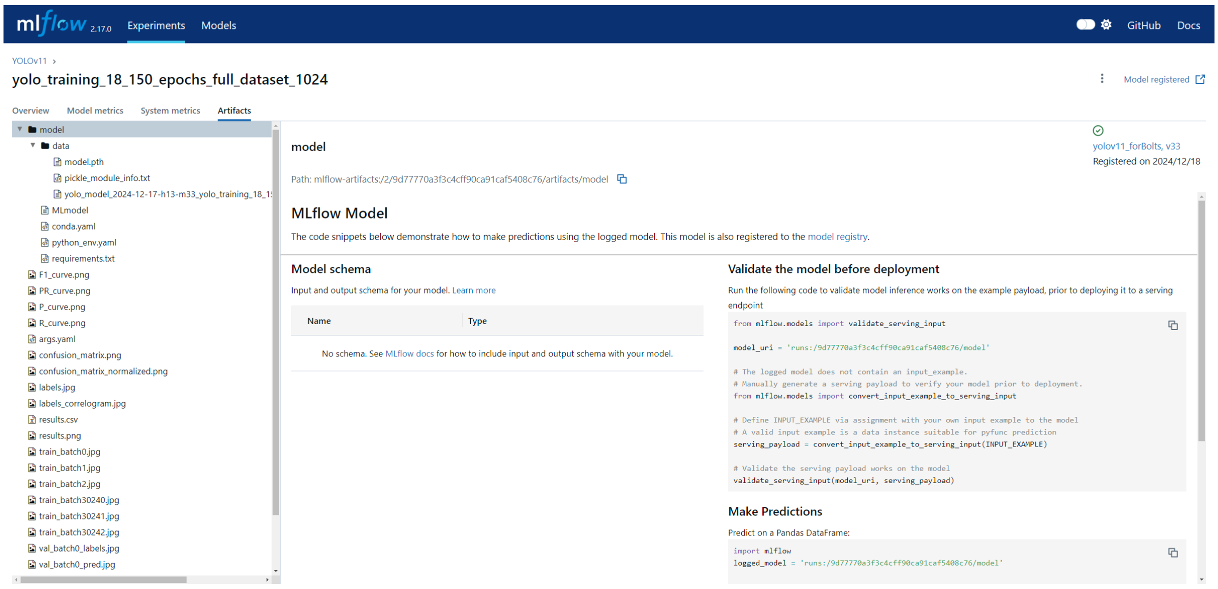


Figure 50: MLFlow YOLOv11 model metrics

The Figure 51 shows the model folder with the saved model and the model artifacts.



yolo_training_18_150_epochs_full_dataset_1024

```
▼ model
  ▼ data
      model.pth
      pickle_module_info.txt
      yolo_model_2024-12-17-h13-m33_yolo_training_18_1
    MLmodel
    conda.yaml
    python_env.yaml
    requirements.txt
  F1_curve.png
  PR_curve.png
  P_curve.png
  R_curve.png
  args.yaml
  confusion_matrix.png
  confusion_matrix_normalized.png
  labels.jpg
  labels_correlogram.jpg
  results.csv
  results.png
  train_batch0.jpg
  train_batch1.jpg
  train_batch2.jpg
  train_batch30240.jpg
  train_batch30241.jpg
  train_batch30242.jpg
  val_batch0_labels.jpg
  val_batch0_pred.jpg
```

model

Path: mlflow-artifacts:/2/9d77770a3f3c4cff90ca91caf5408c76/artifacts/model

yolov11_forBolts, v33
Registered on 2024/12/18

MLflow Model

The code snippets below demonstrate how to make predictions using the logged model. This model is also registered to the model registry.

Model schema

Input and output schema for your model. Learn more

Name	Type
No schema. See MLflow docs for how to include input and output schema with your model.	

Validate the model before deployment

Run the following code to validate model inference works on the example payload, prior to deploying it to a serving endpoint

```
from mlflow.models import validate_serving_input

model_uri = 'runs:/9d77770a3f3c4cff90ca91caf5408c76/model'

# The logged model does not contain an input_example.
# Manually generate a serving payload to verify your model prior to deployment.
from mlflow.models import convert_input_example_to_serving_input

# Define INPUT_EXAMPLE via assignment with your own input example to the model
# A valid input example is a data instance suitable for pyfunc prediction
serving_payload = convert_input_example_to_serving_input(INPUT_EXAMPLE)

# Validate the serving payload works on the model
validate_serving_input(model_uri, serving_payload)
```

Make Predictions

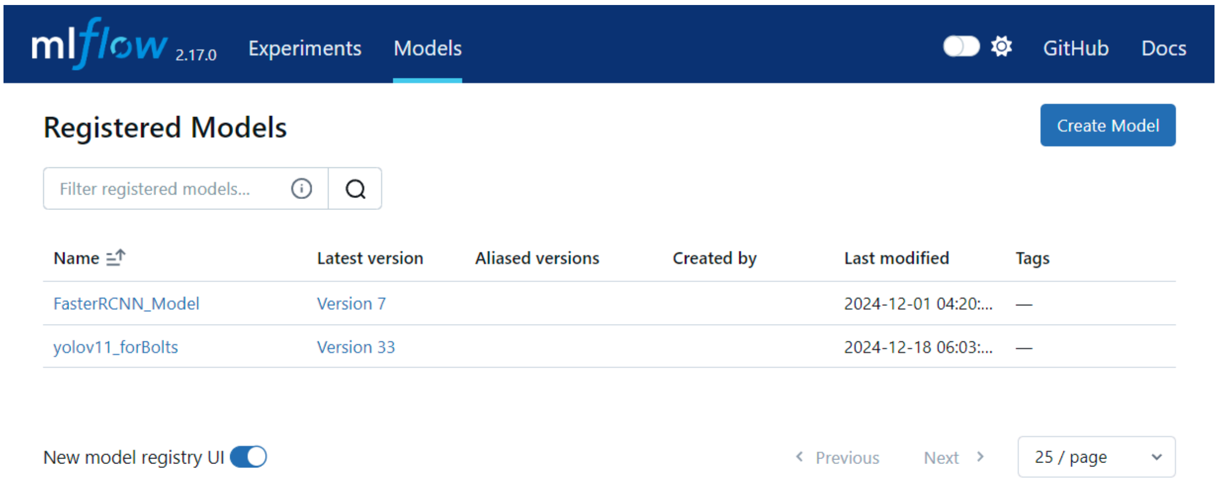
Predict on a Pandas DataFrame:

```
import mlflow
logged_model = 'runs:/9d77770a3f3c4cff90ca91caf5408c76/model'
```

Figure 51: MLFlow YOLOv11 model folder

II 5.2.3.1 MLFlow Model Registry

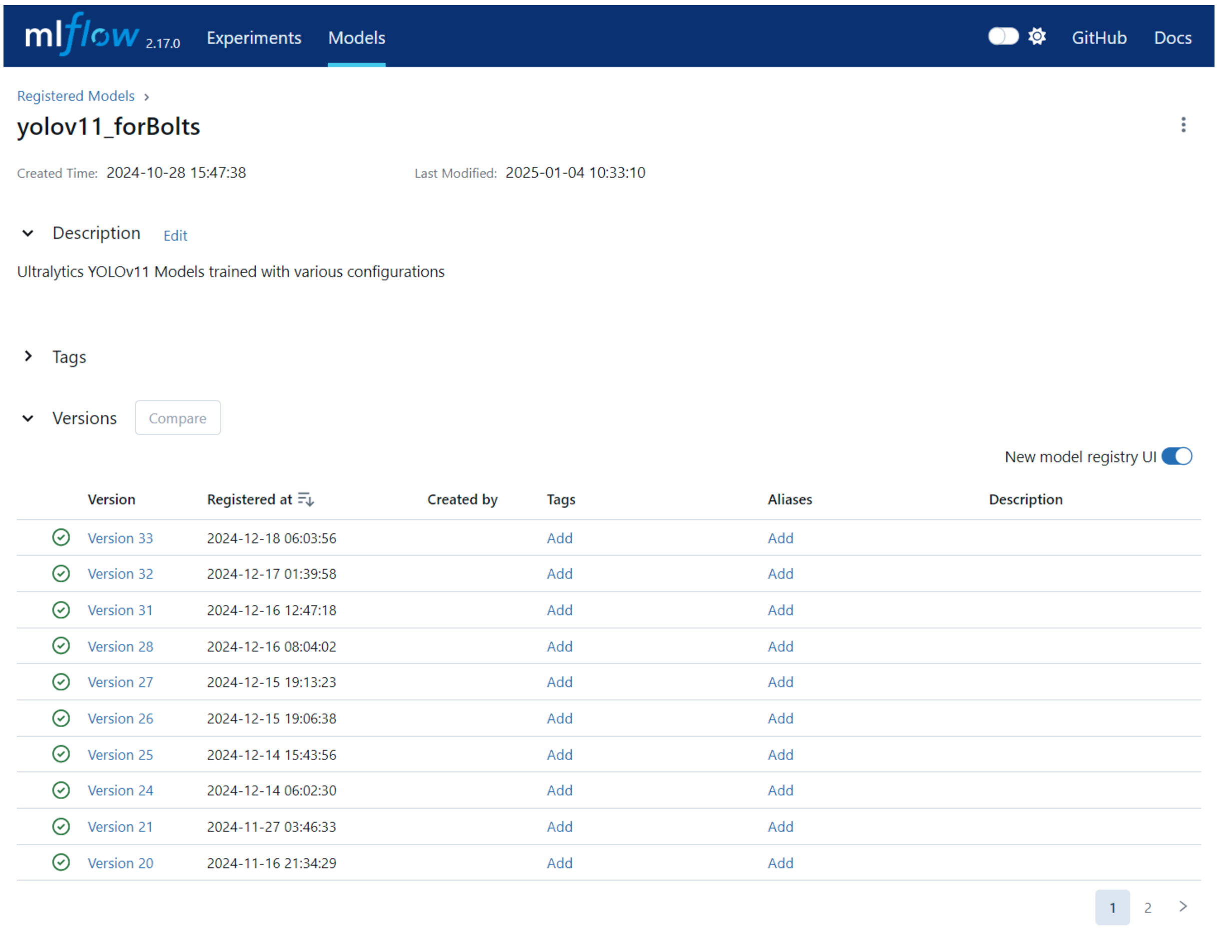
The models are also registered as versions in the MLFlow registry. The following screenshots show the registered models in the MLFlow registry.



Registered Models

Name	Latest version	Aliased versions	Created by	Last modified	Tags
FasterRCNN_Model	Version 7			2024-12-01 04:20:...	—
yolov11_forBolts	Version 33			2024-12-18 06:03:...	—

Figure 52: MLFlow Registered Models for YOLO & Faster R-CNN



yolov11_forBolts

Created Time: 2024-10-28 15:47:38 Last Modified: 2025-01-04 10:33:10

Description

Ultralytics YOLOv11 Models trained with various configurations

Versions

Version	Registered at	Created by	Tags	Aliases	Description
Version 33	2024-12-18 06:03:56		Add	Add	
Version 32	2024-12-17 01:39:58		Add	Add	
Version 31	2024-12-16 12:47:18		Add	Add	
Version 28	2024-12-16 08:04:02		Add	Add	
Version 27	2024-12-15 19:13:23		Add	Add	
Version 26	2024-12-15 19:06:38		Add	Add	
Version 25	2024-12-14 15:43:56		Add	Add	
Version 24	2024-12-14 06:02:30		Add	Add	
Version 21	2024-11-27 03:46:33		Add	Add	
Version 20	2024-11-16 21:34:29		Add	Add	

Figure 53: MLFlow Registered Models for YOLO & Faster R-CNN

More screenshots of the MLFlow models can be found in the appendix Chapter III 4.1.

II 5.3 Inference API Implementation

This section delves into the practical implementation of the Inference API, detailing the workflow of its critical endpoints and internal components.

II 5.3.1 /models Endpoint

The `/models` endpoint provides a list of all available models registered in MLFlow. This endpoint does not perform caching or downloading of models; it only retrieves metadata about the models from MLFlow and returns it to the client. The purpose of this endpoint is to allow users to view available models and their versions before making predictions. Code-Fragment 12 shows the implementation of this endpoint.

Implementation of the `/models` endpoint in the Inference API.

```python
@router.get("/models", response_model=List[ModelDescription])
async def list_models(
    model_service: callable = Depends(get_model_service)
):
    try:
        models = model_service()
        logger.info(f"Found {len(models)} interfaces")
        return models
    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=f"An error occurred while fetching interfaces: {str(e)}")

```

Code-Fragment 12: Implementation of the `/models` endpoint in the Inference API.

This design ensures that only metadata is fetched at this stage, avoiding unnecessary downloads and ensuring efficient use of resources.

II 5.3.2 /predict Endpoint

The `/predict` endpoint is the core of the Inference API, responsible for handling inference requests. Unlike the `/models` endpoint, it checks if the required model is already cached. If not, the model is downloaded from MLFlow, cached locally, and used for inference. Upon receiving a request, it performs the following steps: The high-level workflow for the `/predict` endpoint is illustrated below in Figure 54 and the implementation of the endpoint is shown in Code-Fragment 13.

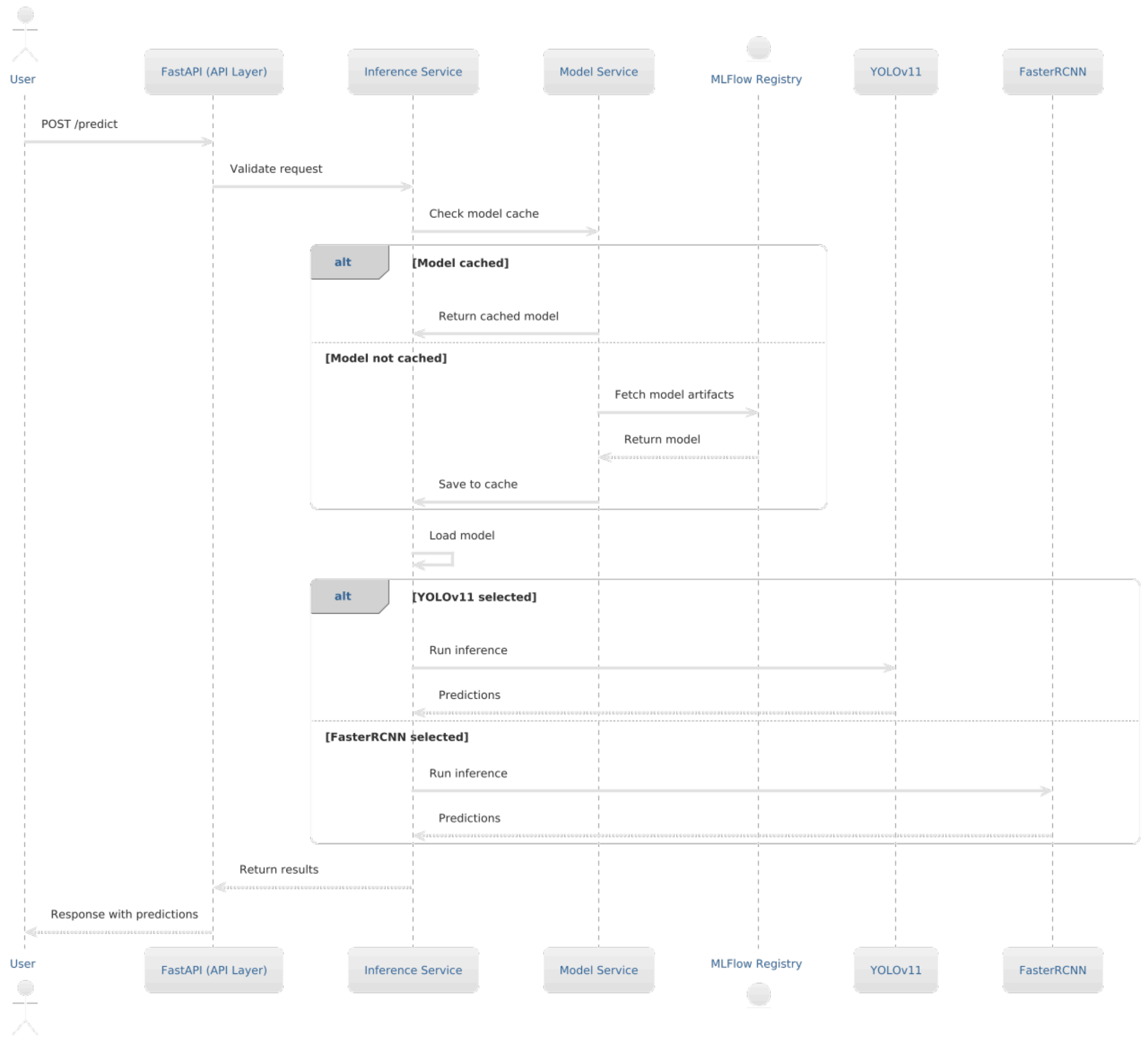


Figure 54: High-level workflow for the /predict endpoint.

1. Accepts an uploaded image and prediction options, such as model name, version, and slicing parameters.
2. Preprocesses the image, slicing it into smaller tiles based on the provided options.
3. Checks if the specified model is cached. If not, downloads it from MLFlow and caches it locally.
4. Runs inference on the image slices using the loaded model.
5. Postprocesses the results, applying techniques like non-maximum suppression (NMS) to filter overlapping predictions.
6. Returns the processed predictions to the client.

Implementation of the `/predict` endpoint in the Inference API.

```python
@router.post("/predict", response_model=PredictionResults)
async def predict(
    image: UploadFile = File(...),
    options: PredictionOptions = Depends(),
    interface_service: callable = Depends(get_interface_service)
):
    try:
        temp_image_path = save_upload_file(image)
        predictions = interface_service(temp_image_path, options)
        remove_file(temp_image_path)

        return PredictionResults(predictions=predictions or [])
    except Exception as e:
        raise HTTPException(status_code=500, detail=f"Prediction failed: {str(e)}")
```

Code-Fragment 13: Implementation of the `/predict` endpoint in the Inference API.

II 5.3.2.1 Image Slicing in Prediction Process

An integral part of the prediction process is slicing the input images into smaller tiles for more effective detection of smaller objects. The slicing parameters are configurable through `PredictionOptions`, allowing customization of slice dimensions and overlap ratios.

The slicing process is implemented using the `sahi.predict` library, as shown in Code-Fragment 14:

Implementation of the slicing process in the prediction pipeline.

```python
from sahi.predict import get_sliced_prediction

results = get_sliced_prediction(
    image_path,
    self.model,
    slice_height=options.slice_height,
    slice_width=options.slice_width,
    overlap_height_ratio=options.overlap_height_ratio,
    overlap_width_ratio=options.overlap_width_ratio,
)
```

Code-Fragment 14: Implementation of the slicing process in the prediction pipeline.

The SAHI library slices the image as illustrated in Figure 55. The tile is moving over the image and detecting objects in each slice. The results are then aggregated to provide the final predictions.

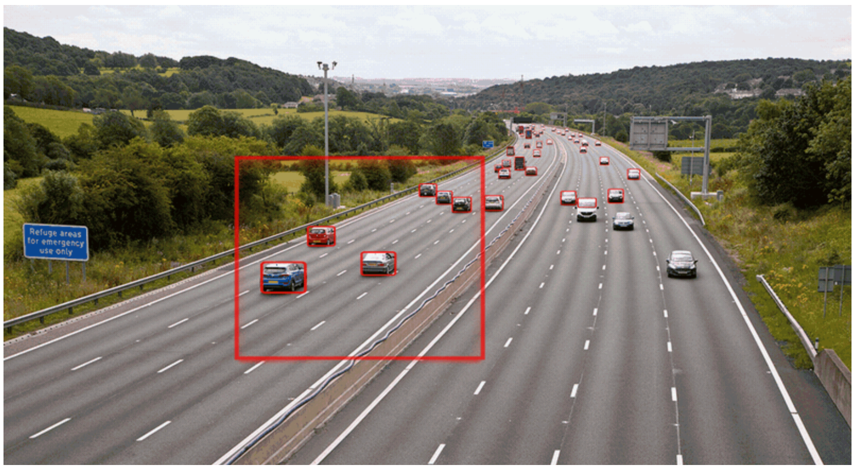


Figure 55: SAHI Slicing Example with Cars in the Distance. [29]

This implementation ensures that even smaller objects are detected efficiently by dividing the image into manageable slices, processing them individually, and then aggregating the results.

II 5.3.3 Model Loading and Caching

Model management is a critical aspect of the Inference API. The system dynamically fetches models from MLFlow and caches them locally for future use (Figure 56). This mechanism ensures low latency and reduces network overhead. The caching process occurs during prediction: it first checks whether the required model is already cached. If not, the model is downloaded, stored locally, and reused for subsequent predictions.

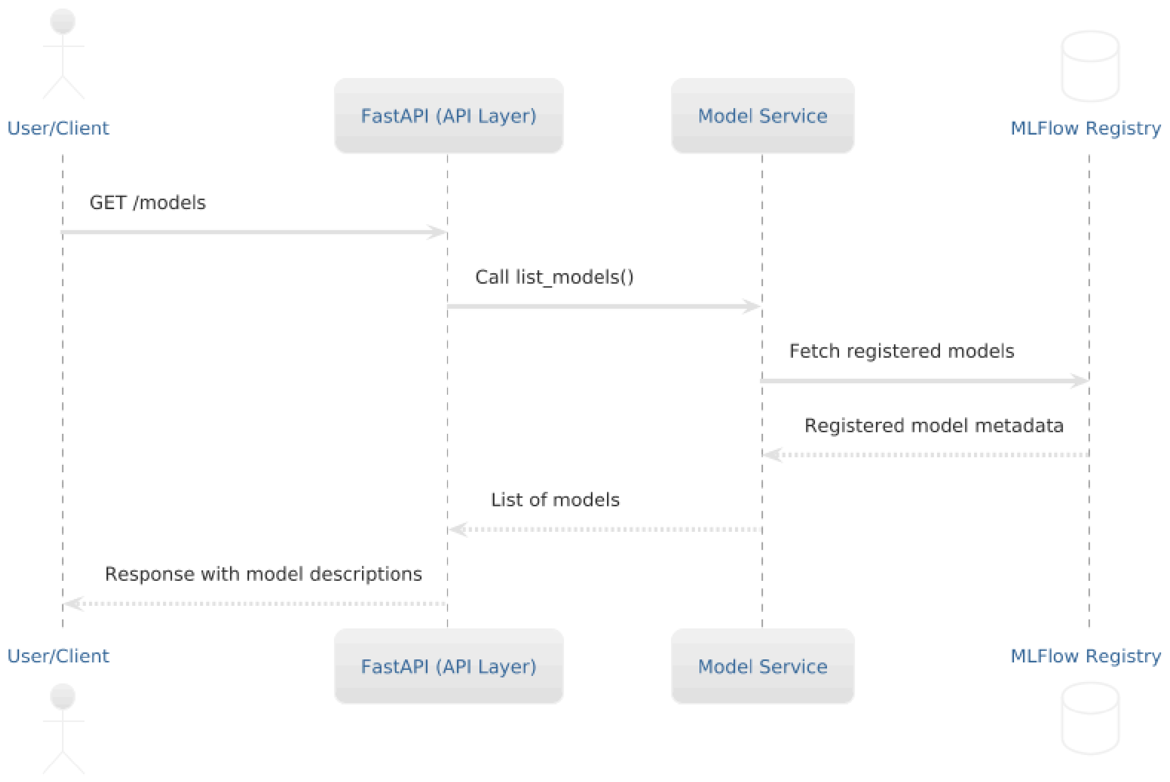


Figure 56: High-level workflow for the /predict endpoint.

The `get_cached_model` function in Code-Fragment 15 illustrates this process:

Implementation of the `get_cached_model` function for caching machine learning models.

```python
def get_cached_model(model_name: str, model_version: str, mlflow_client=mlflow) -> str:
    local_model_dir = os.path.join(MODEL_DIR, model_name, model_version)

    if os.path.exists(local_model_dir):
        return find_model_file(local_model_dir)

    mlflow_model_uri = get_model_uri(model_name, model_version)
    tmp_model_dir = mlflow_client.artifacts.download_artifacts(mlflow_model_uri)
    os.makedirs(local_model_dir, exist_ok=True)

    for item in os.listdir(tmp_model_dir):
        source_path = os.path.join(tmp_model_dir, item)
        target_path = os.path.join(local_model_dir, item)
        shutil.move(source_path, target_path)

    shutil.rmtree(tmp_model_dir)
    return find_model_file(local_model_dir)
```

Code-Fragment 15: Implementation of the `get_cached_model` function for caching machine learning models.

II 5.3.4 Extensibility with New Models

The implementation allows easy addition of new models by adhering to the `InferenceEngine` interface. For example, adding a new model involves implementing the `load_model` and `run_inference` methods. This modular approach ensures that inference for different model architectures can be implemented seamlessly, enabling the system to adapt to emerging technologies.

The `InferenceEngine` base class provides the required structure (Code-Fragment 16):

Definition of the `InferenceEngine` abstract base class.

```python
from abc import ABC, abstractmethod
from src.api.schemas import PredictionOptions

class InferenceEngine(ABC):
    @abstractmethod
    def load_model(self, model_path: str):
        pass

    @abstractmethod
    def run_inference(self, image_path: str, options: PredictionOptions):
        pass
```

Code-Fragment 16: Definition of the `InferenceEngine` abstract base class.

A concrete implementation for Faster R-CNN would extend this class (Code-Fragment 17):

Implementation of the `FasterRCNNInference` class for object detection using Faster R-CNN.

```python
class FasterRCNNInference(InferenceEngine):
def load_model(self, model_path: str):
    checkpoint = torch.load(model_path, map_location=self.device)
    self.model = fasterrcnn_resnet50_fpn(weights=None)
    self.model.load_state_dict(checkpoint)
    self.model.to(self.device)
    self.model.eval()

def run_inference(self, image_path: str, options: PredictionOptions):
    image = Image.open(image_path).convert("RGB")
    image_tensor = F.to_tensor(image).unsqueeze(0).to(self.device)
    predictions = self.model(image_tensor)[0]
    return process_predictions(predictions, options)
```

Code-Fragment 17: Implementation of the `FasterRCNNInference` class for object detection using Faster R-CNN.

II 5.3.5 Dependency Injection

Dependency injection is utilized in the endpoints to allow components to be swapped easily. This design ensures flexibility and simplifies testing by decoupling the API from specific implementations of services. For instance, the `model_service` and `interface_service` dependencies are injected as follows in Code-Fragment 18:

Dependency injection for `model_service` and `interface_service`.

```python
# Dependency injection for model_service and interface_service
def get_model_service():
    return lambda: default_list_models(client=mlflow_client)

def get_interface_service():
    return default_run_inference
```

Code-Fragment 18: Dependency injection for `model_service` and `interface_service`.

This approach makes the API adaptable to changes in service implementations, further enhancing maintainability and scalability.

II 5.3.6 Configuration

The API is configurable using an `.env` file, enabling flexibility for deployment across different environments. Key settings such as MLFlow tracking URI, username, and password are defined in the environment file depicted in Code-Fragment 19:

Environment variable configuration for MLFlow tracking.

```
MLFLOW_TRACKING_URI="https://mlflow-ba.infs.ch"
MLFLOW_TRACKING_USERNAME="username"
MLFLOW_TRACKING_PASSWORD="token"
```

Code-Fragment 19: Environment variable configuration for MLFlow tracking.

This configuration approach ensures that sensitive information is not hardcoded into the application, enhancing security and maintainability.

II 5.4 Backend API Implementation

The backend of the BoltFinder project is implemented using **ExpressJS**, a lightweight web application framework for Node.js. The backend is written in **TypeScript** to ensure type safety and scalability. Acting as the central orchestrator, it connects the frontend, inference API, database, and auxiliary services such as the Cantaloupe server.

The API is organized into various endpoint categories that handle different functionalities. These include managing image uploads, interacting with prediction models, tracking and updating jobs, and generating climbing route data and visualizations. Below in Table 20 is an overview of the endpoint categories and their purposes:

Endpoint Group	Description
`/api/upload`	Handles image uploads and storage.
`/api/inference`	Provides model listing and prediction capabilities.
`/api/jobs`	Tracks and manages job statuses.
`/api/topology`	Manages topology generation.

Table 20: Endpoint Groups and Descriptions

II 5.4.1 Server Setup

The backend API is designed with middleware to handle logging, request validation, error handling, and routing. File uploads are processed using **Multer**, and static files are served from the `public` directory.

The backend API communicates with a PostgreSQL database for persistent data storage, leveraging **TypeORM** for object-relational mapping. Additionally, it interacts with the Inference API to handle model-based predictions, and the Cantaloupe server is used for serving and processing image-related tasks.

II 5.4.2 Database and Entities

The backend of the BoltFinder project is built on a PostgreSQL database modeled using **TypeORM**. This ensures clearly defined relationships between entities, enabling seamless interaction with the data layer. The database serves as the foundation for core functionalities such as image uploads, prediction management, and climbing route generation.

The data model revolves around seven key tables: `image`, `job`, `model`, `prediction`, `route`, `prediction_options` and `route_prediction`. Each entity plays a distinct role, working together to manage data efficiently.

To store and manage data, the `Image` entity records metadata about uploaded images, such as filenames, dimensions, and file paths. Each image is associated with one or more `Job` entities. Machine learning models are represented by the `Model` entity, which stores models uniquely identified by name and version. Configuration details for prediction tasks, such as confidence thresholds and slicing parameters, are stored in the `PredictionOptions` entity and linked to jobs.

II 5.4.2.1 Detailed Entity Relationships

The diagram in Figure 57 below provides a closer look at how predictions and routes are managed.

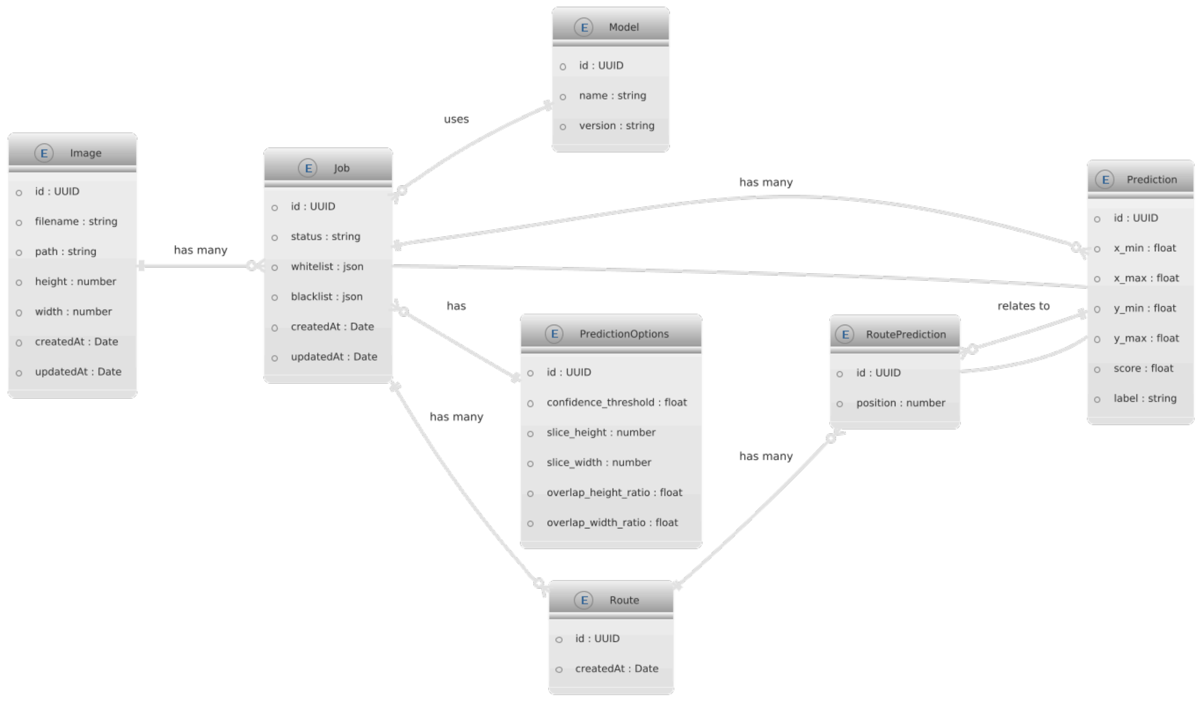


Figure 57: Detailed Entity Relationships for Backend Data

The `Prediction` entity stores the output of the inference process, including bounding box coordinates, labels, and confidence scores. These predictions are linked to climbing routes through the `Route` and `RoutePrediction` entities. A `Route` represents a calculated climbing path, while a `RoutePrediction` links specific predictions to their positions in the route. This structure allows for flexible route management and ensures each route is directly tied to the underlying predictions.

II 5.4.2.2 Route Structure

The example Figure 58 demonstrates how a job can manage multiple routes. For instance:

- **Job 1** is associated with two routes: `Route 1` and `Route 2`.
- **Route 1** begins with a prediction labeled "Anchor" and proceeds to a prediction labeled "Bolt".
- **Route 2** also starts at the same anchor but ends at a different bolt.

Each route is defined as a sequence of `RoutePredictions`, which link specific `Predictions` to their positions in the path.

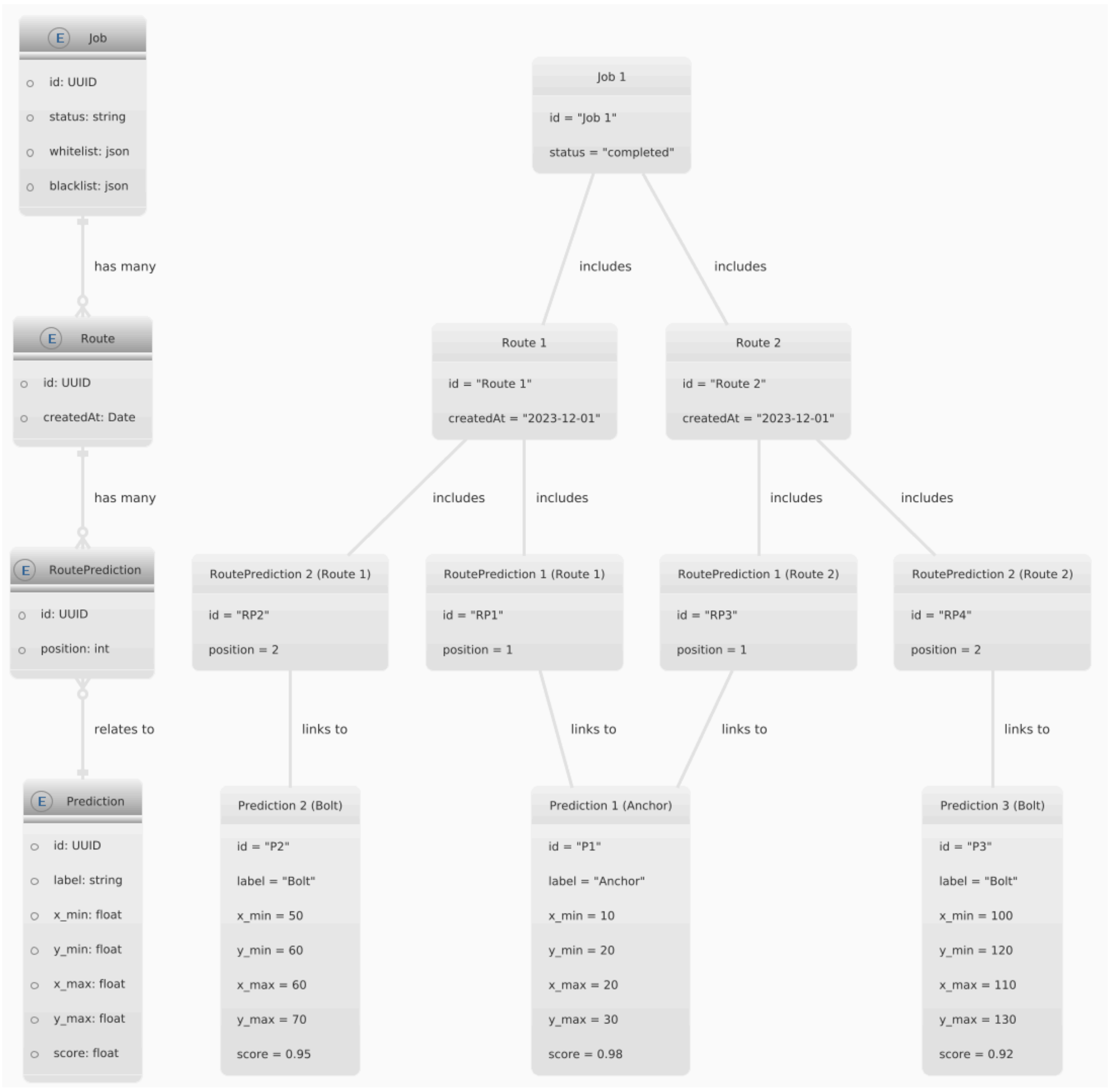


Figure 58: Route Example: How Jobs, Routes, and Predictions are Structured

This design enables dynamic and precise route generation. If predictions change, the backend can easily recalculate routes or add new ones based on updated data. By organizing the data model around these key entities, the backend achieves a flexible and scalable architecture that supports advanced functionalities such as real-time prediction updates, and dynamic route recalculation.

II 5.4.3 API Endpoints

The backend provides several endpoints that facilitate various operations, such as image management, job tracking, prediction processing, and climbing route generation. These endpoints are organized into logical route groups. The route groups make it easier to navigate the API. Below in Table 21 is a brief overview of the API endpoints and the next sections explain the API in detail including sequence diagrams that illustrate how each endpoint processes requests.

Route Group	**Endpoints**
`/api/upload`	`POST /api/upload`: Upload an image to the server, `GET /api/upload/list`: List uploaded images.
`/api/inference`	`GET /api/inference/models`: List available models, `POST /api/inference/predict`: Start a prediction job,
`/api/jobs`	`GET /api/jobs`: List all jobs, `GET /api/jobs/:jobId`: Get details of a specific job, `DELETE /api/jobs/:jobId`: Delete a specific job, `POST /api/jobs/update`: Update predictions and routes for a job, `POST /api/jobs/recalculate`: Recalculate routes for a job.
`/api/topology`	`POST /api/topology/download/topology/:jobId`: Download a topology map, `POST /api/topology/download/voronoi/:jobId`: Download a Voronoi diagram.

Table 21: Route Groups and Associated Endpoints

Each route group is described in detail, along with its functionality and workflows in the following sections.

II 5.4.3.1 Upload Routes (`/api/upload`)

The `/api/upload` route group handles the management of uploaded images, enabling users to upload new images and retrieve information about stored files.

II 5.4.3.1.1 `/api/upload` (POST)

This endpoint accepts image uploads, validates them, and saves them to the server (Figure 59). When a client sends a file, the backend sanitizes the filename, extracts metadata (such as dimensions), and stores the file in the `public/uploads` directory. The image metadata is also saved in the database for future reference.

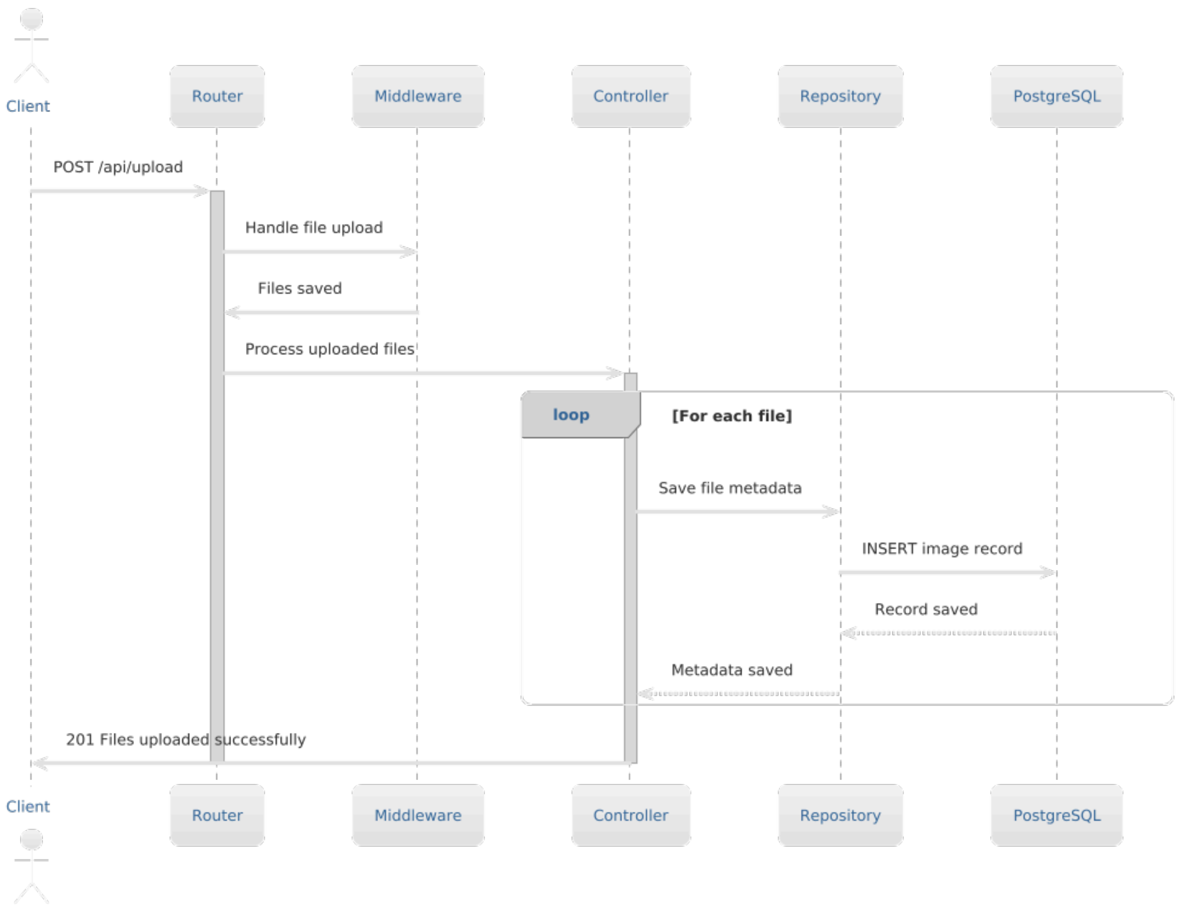


Figure 59: Sequence Diagram for `/api/upload`

The sequence diagram in Figure 59 above shows the steps for processing an image upload:
1. The client sends an image file to the server via `POST /api/upload`.
2. The backend validates the file and saves it in the uploads directory.
3. Metadata is extracted and stored in the database.
4. The server responds to the client with a confirmation of successful upload.

II 5.4.3.1.2 `/api/upload/list` (GET)

This endpoint retrieves a list of all uploaded images stored on the server. The backend queries the database to fetch metadata for each uploaded image, including filenames, file paths, and dimensions.

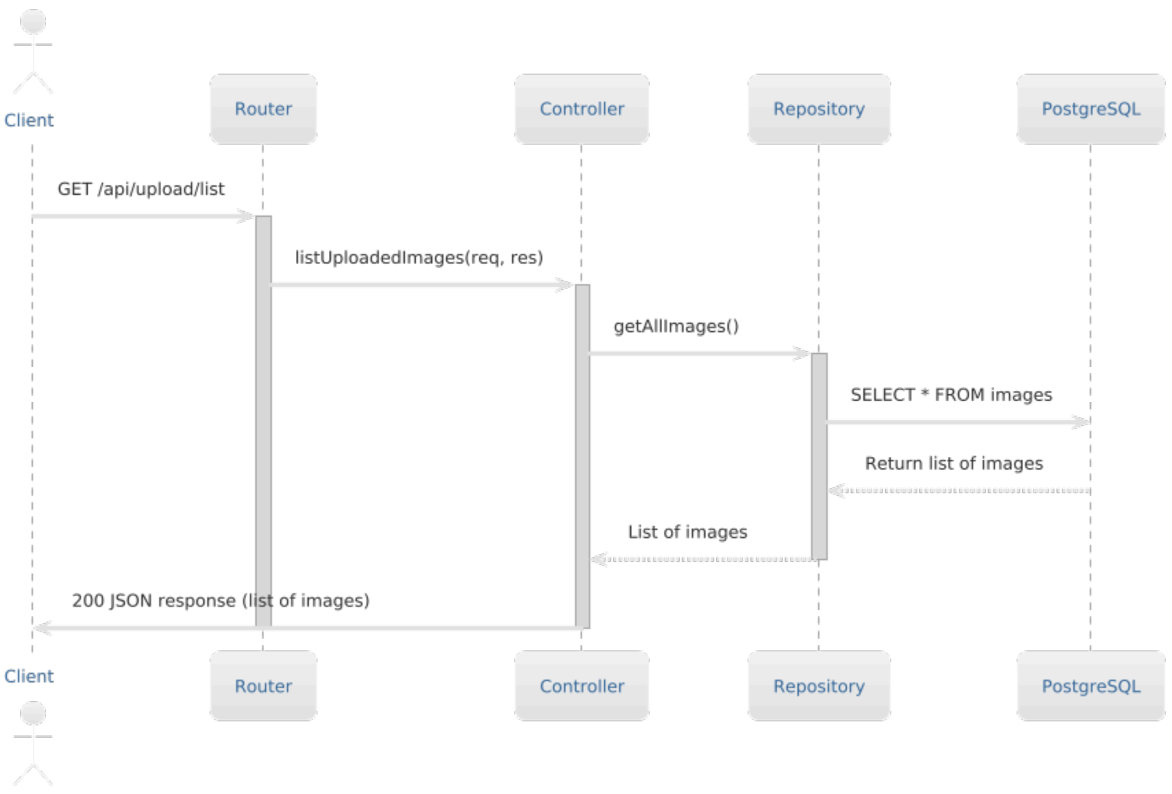


Figure 60: Sequence Diagram for `/api/upload/list`

As shown in the sequence diagram in Figure 60, the workflow involves:

1. The client requests a list of uploaded images via `GET /api/upload/list`.
2. The backend queries the database for all image metadata.
3. The results are returned to the client as a JSON response.

II 5.4.3.2 Inference Routes (`/api/inference`)

The `/api/inference` route group enables interaction with machine learning models and facilitates prediction tasks. These routes abstract the complexity of the inference API and provide a simplified interface to the client.

II 5.4.3.2.1 `/api/inference/models` (GET)

This endpoint retrieves a list of all available models stored in the backend database. The models are synchronized periodically with the inference API and include details such as the model name and version.

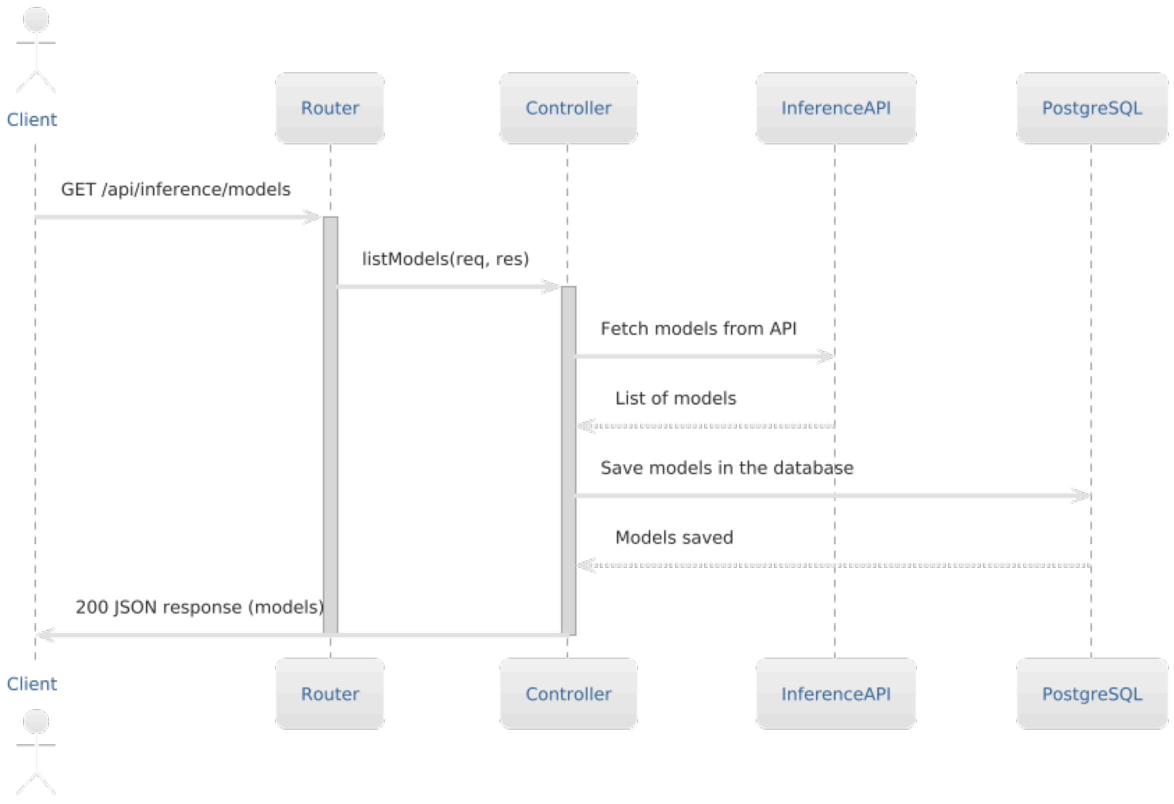


Figure 61: Sequence Diagram for `/api/inference/models`

The sequence diagram in Figure 61 illustrates the process:
1. The client sends a `GET /api/inference/models` request.
2. The backend queries the database for synchronized models.
3. A list of models is returned to the client.

II 5.4.3.2.2 `/api/inference/predict` (POST)

This endpoint allows clients to submit image IDs for prediction. Clients must specify a model and prediction options (e.g., confidence threshold). Once validated, the backend creates prediction jobs for the submitted images, processes the jobs in the background, and immediately returns job IDs for tracking.

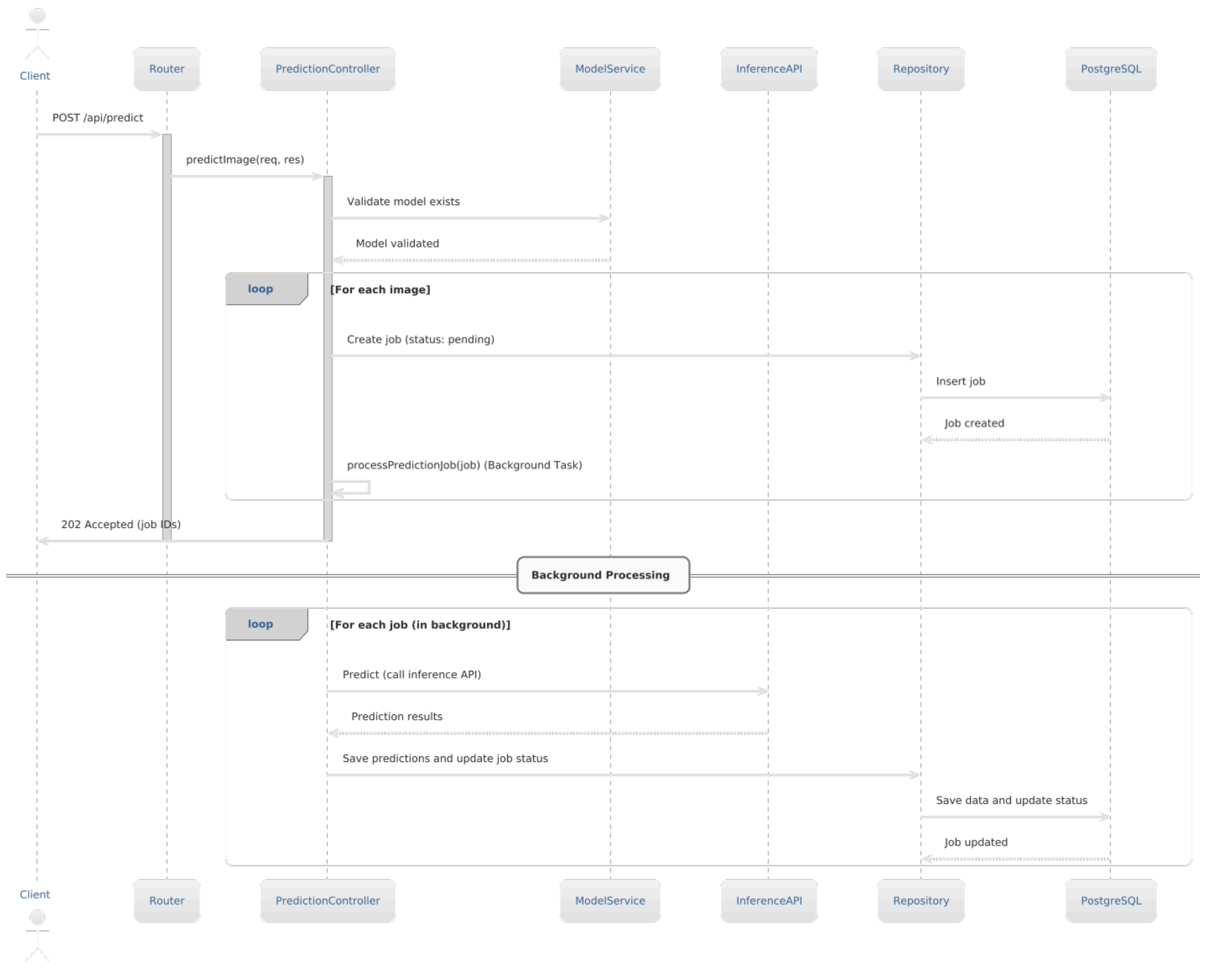


Figure 62: Sequence Diagram for `/api/inference/predict`

In the sequence diagram in Figure 62:

1. The client submits a `POST /api/inference/predict` request with image IDs and options.
2. The backend validates the request and creates jobs in the database.
3. The prediction process is triggered in the background.
4. The client receives a response with job IDs.

II 5.4.3.3 Job Routes (`/api/jobs`)

The `/api/jobs` route group manages prediction jobs, enabling users to track, update, or delete jobs and recalculate routes.

II 5.4.3.3.1 `/api/jobs` (GET)

This endpoint retrieves a list of all jobs stored in the database, including their statuses and metadata.

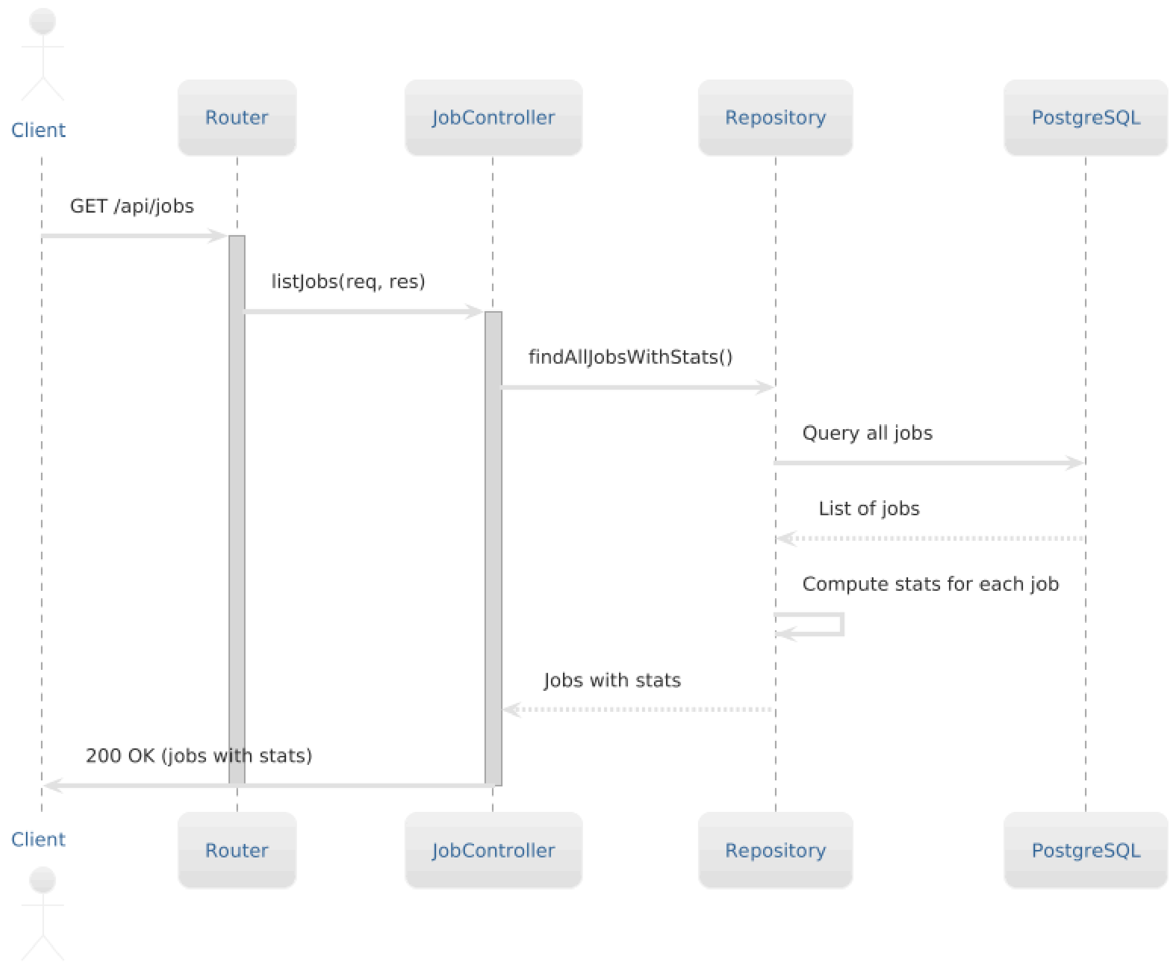


Figure 63: Sequence Diagram for `/api/jobs`

In the diagram above in Figure 63:
1. The client sends a `GET /api/jobs` request.
2. The backend queries the database for all jobs.
3. The list of jobs is returned as a JSON response.

II 5.4.3.3.2 `/api/jobs/:jobId` (GET)

This endpoint fetches detailed information about a specific job, including its predictions and associated routes.

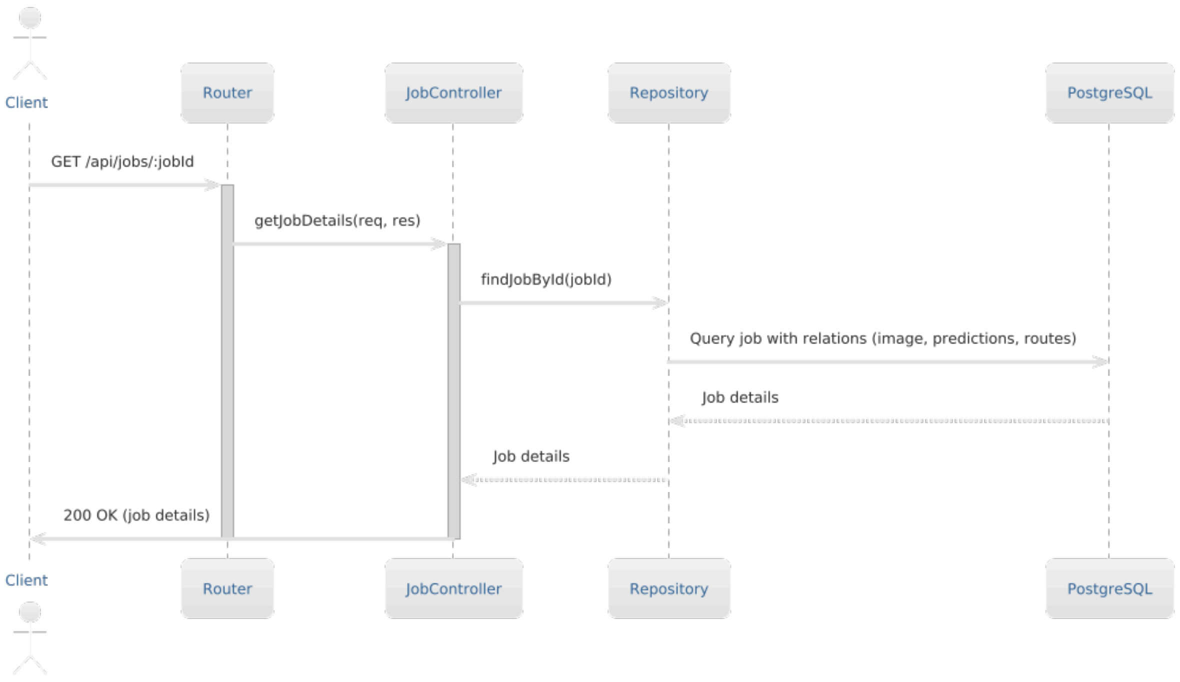


Figure 64: Sequence Diagram for `/api/jobs/:jobId`

The sequence diagram in Figure 64 shows:

1. The client sends a `GET /api/jobs/:jobId` request.
2. The backend queries the database for details of the specified job.
3. The job data is returned as a JSON response.

II 5.4.3.3.3 `/api/jobs/:jobId` (DELETE)

This endpoint deletes a specific job from the database (Figure 65). The backend also removes associated predictions and routes.

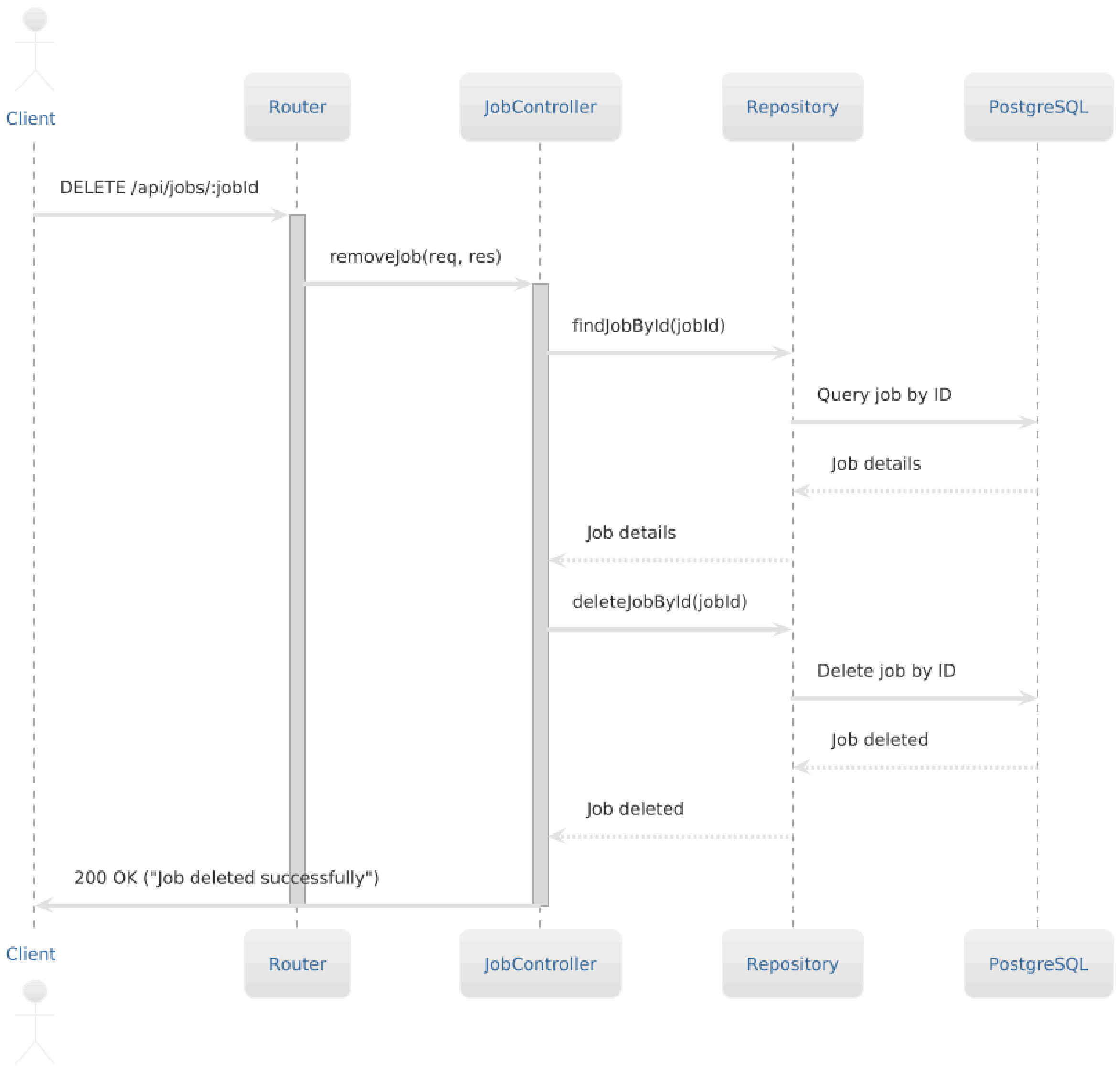


Figure 65: Sequence Diagram for `/api/jobs/:jobId` (DELETE)

II 5.4.3.3.4 `/api/jobs/update` (POST)

This endpoint updates the predictions and routes for a specific job (Figure 66). Users can submit new predictions, modify existing ones, or remove outdated ones.

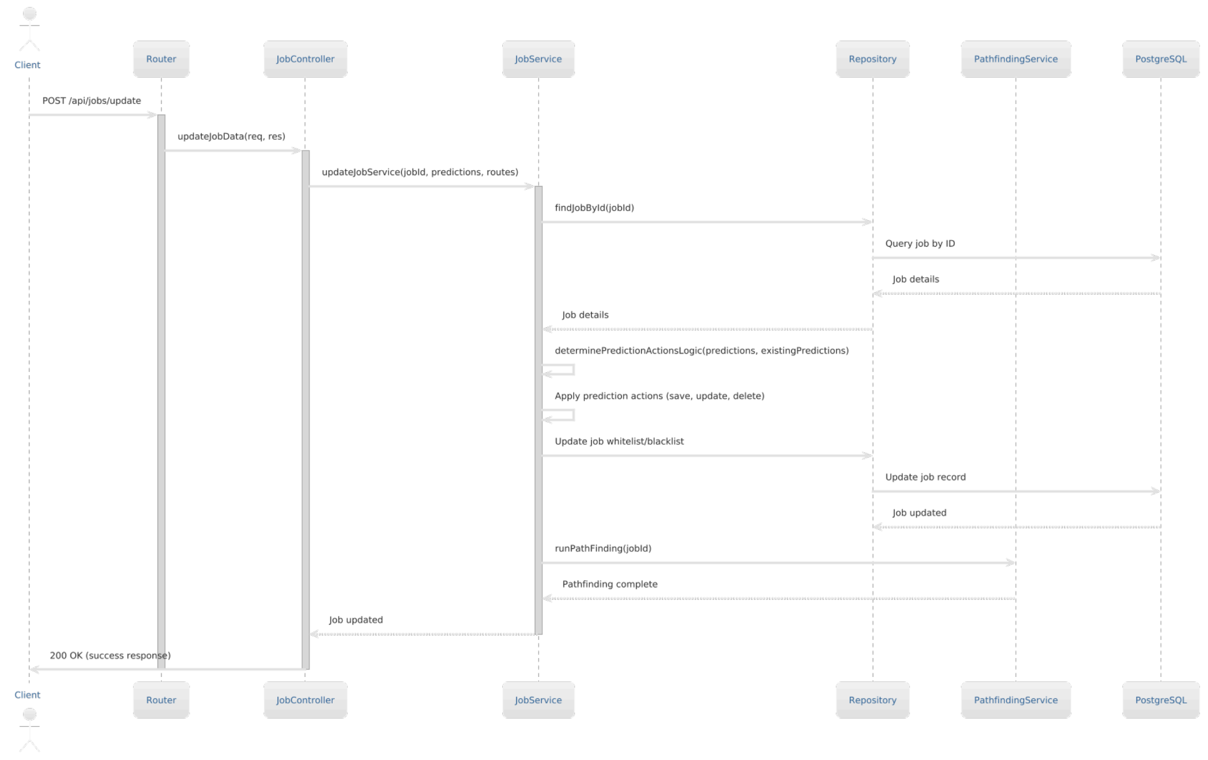


Figure 66: Sequence Diagram for `/api/jobs/update`

II 5.4.3.3.5 `/api/jobs/recalculate` (POST)

This endpoint resets the whitelist and blacklist for a job and recalculates its routes (Figure 67).

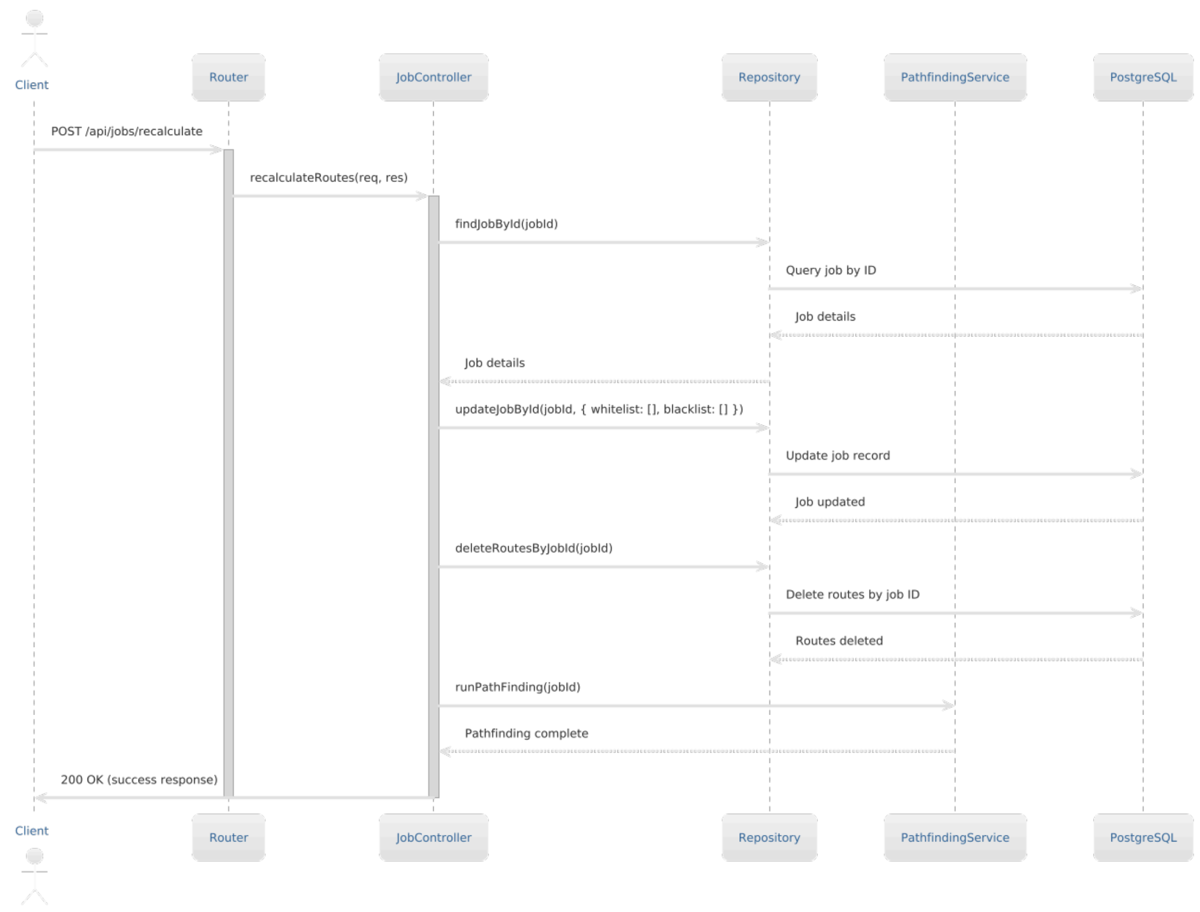


Figure 67: Sequence Diagram for `/api/jobs/recalculate`

II 5.4.3.4 Topology Routes (`/api/topology`)

The `/api/topology` route group generates visualizations of climbing routes.

II 5.4.3.4.1 `/api/topology/download/topology/:jobId` (POST)

Generates a topology map for the specified job and provides it as a downloadable file (Figure 68).

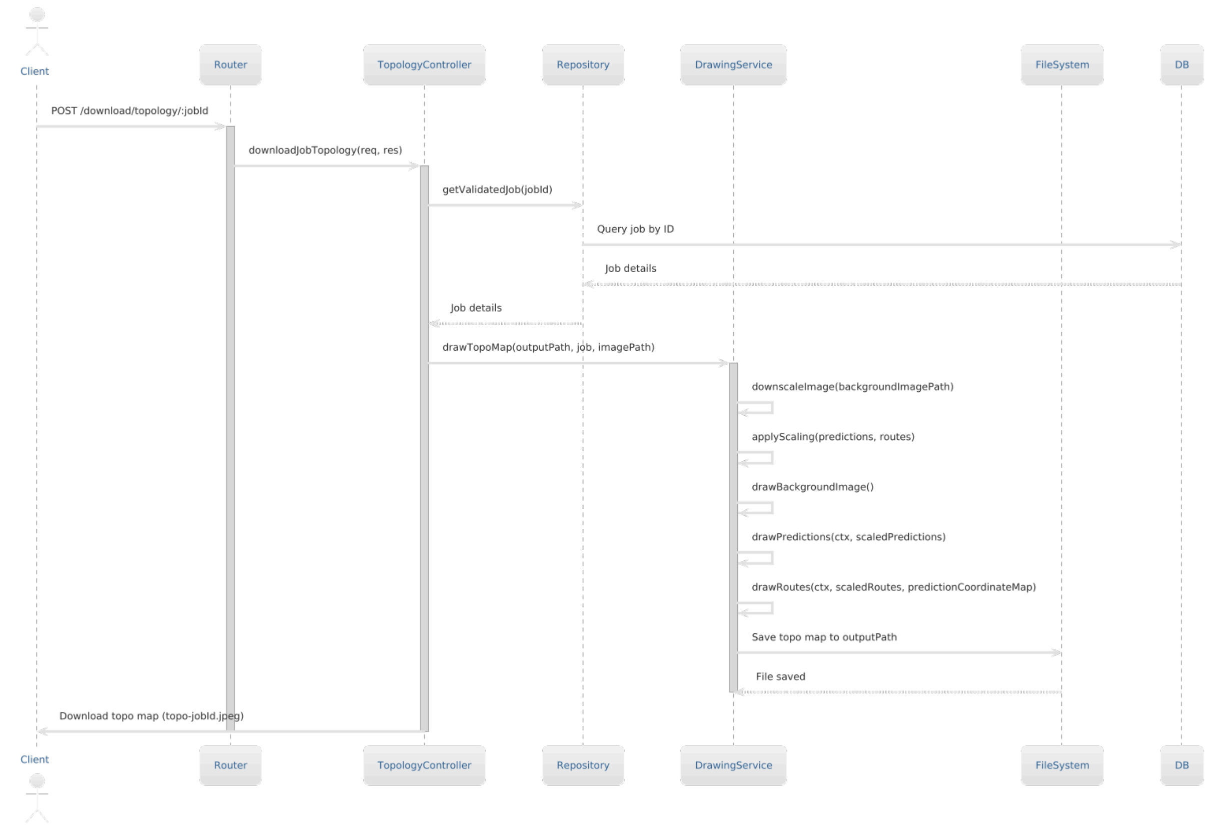

Client
Router
TopologyController
Repository
DrawingService
FileSystem
DB
POST /download/topology/:jobId
downloadJobTopology(req, res)
getValidatedJob(jobId)
Query job by ID
Job details
Job details
drawTopoMap(outputPath, job, imagePath)
downscaleImage(backgroundImagePath)
applyScaling(predictions, routes)
drawBackgroundImage()
drawPredictions(ctx, scaledPredictions)
drawRoutes(ctx, scaledRoutes, predictionCoordinateMap)
Save topo map to outputPath
File saved
Download topo map (topo-jobId.jpeg)

Figure 68: Sequence Diagram for `/api/topology/download/topology/:jobId`

II 5.4.3.4.2 `/api/topology/download/voronoi/:jobId` (POST)

Generates a Voronoi diagram for the specified job and provides it as a downloadable file (Figure 69).

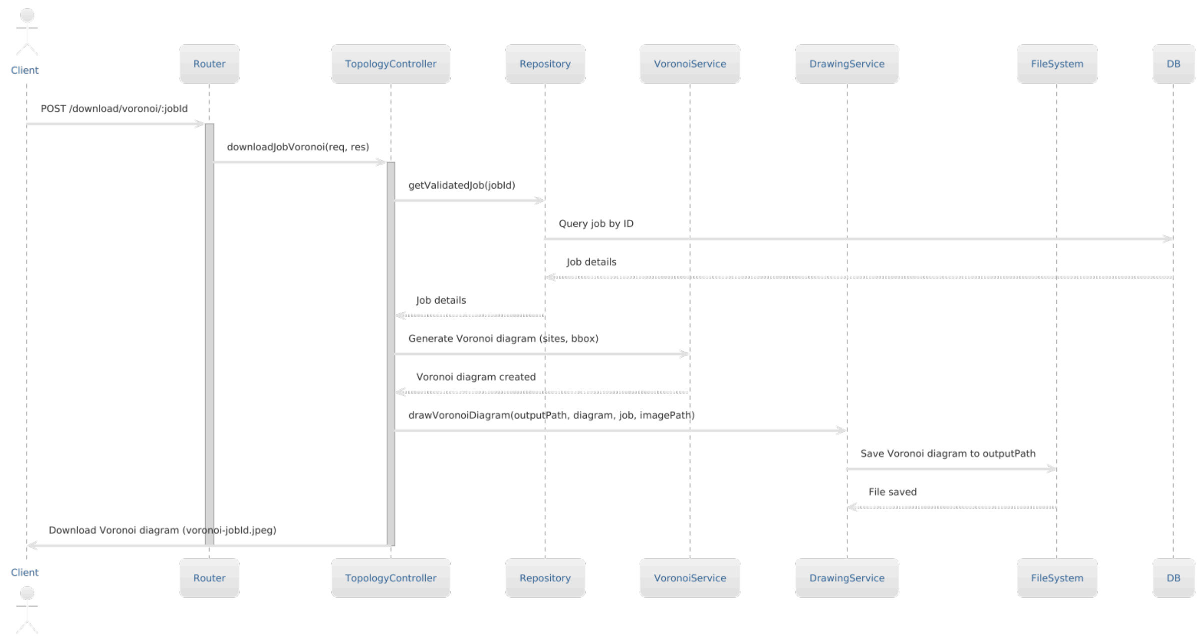

Client
Router
TopologyController
Repository
VoronoiService
DrawingService
FileSystem
DB
POST /download/voronoi/:jobId
downloadJobVoronoi(req, res)
getValidatedJob(jobId)
Query job by ID
Job details
Job details
Generate Voronoi diagram (sites, bbox)
Voronoi diagram created
drawVoronoiDiagram(outputPath, diagram, job, imagePath)
Save Voronoi diagram to outputPath
File saved
Download Voronoi diagram (voronoi-jobId.jpeg)

Figure 69: Sequence Diagram for `/api/topology/download/voronoi/:jobId`

II 5.4.4 Configuration

The backend system for the BoltFinder application relies on a set of configurable environment variables to define its behavior, connections, and runtime properties. These configurations are divided into common settings, PostgreSQL-related variables, application ports, and hostname details.

II 5.4.4.1 Common Configuration

The following environment variables in Code-Fragment 20 are used to define general backend settings:

Common configuration settings for the BoltFinder backend.

```
NODE_ENV=development
DEBUG=true
LOG_LEVEL=debug
ADDRESS=0.0.0.0
SWAGGER_ADDRESS=localhost
POSTGRES_ADDRESS=boltfinder-postgres-db
PORT=3000
RELOAD=true
CHOKIDAR_USEPOLLING=true
MAX_IMAGE_SIZE=100
INFERENCE_API=http://hn601948.ost.ch:8000
```

Code-Fragment 20: Common configuration settings for the BoltFinder backend.

II 5.4.4.2 PostgreSQL Configuration

The following variables in Code-Fragment 21 define the database connection details for the PostgreSQL database used by the backend:

PostgreSQL configuration for the BoltFinder backend.

```
POSTGRES_USER=<user>
POSTGRES_PASSWORD=<pw>
POSTGRES_DB=<db>
```

Code-Fragment 21: PostgreSQL configuration for the BoltFinder backend.

II 5.4.4.3 Application Ports

The backend system uses specific ports for different services, as configured below in Code-Fragment 22:

Port configuration for the BoltFinder application.

```
BACKEND_PORT=3000
CANTALOUPE_PORT=8182
POSTGRES_PORT=5432
```

Code-Fragment 22: Port configuration for the BoltFinder application.

II 5.4.4.4 Hostname

The hostname for the backend is defined as follows in Code-Fragment 23:

Hostname configuration for the backend.

```
HOSTNAME=localhost
```

Code-Fragment 23: Hostname configuration for the backend.

II 5.4.5 Route Finding Algorithm

The Route Finding Algorithm is designed to identify climbing routes on a wall by analyzing the positions of bolts and anchors. The algorithm processes those bolts and anchors, returning a list of routes, each defined as a sequence of bolts leading to a single anchor.

Ideally the algorithm should be able to detect the routes in the following by hand annotated image Figure 70:

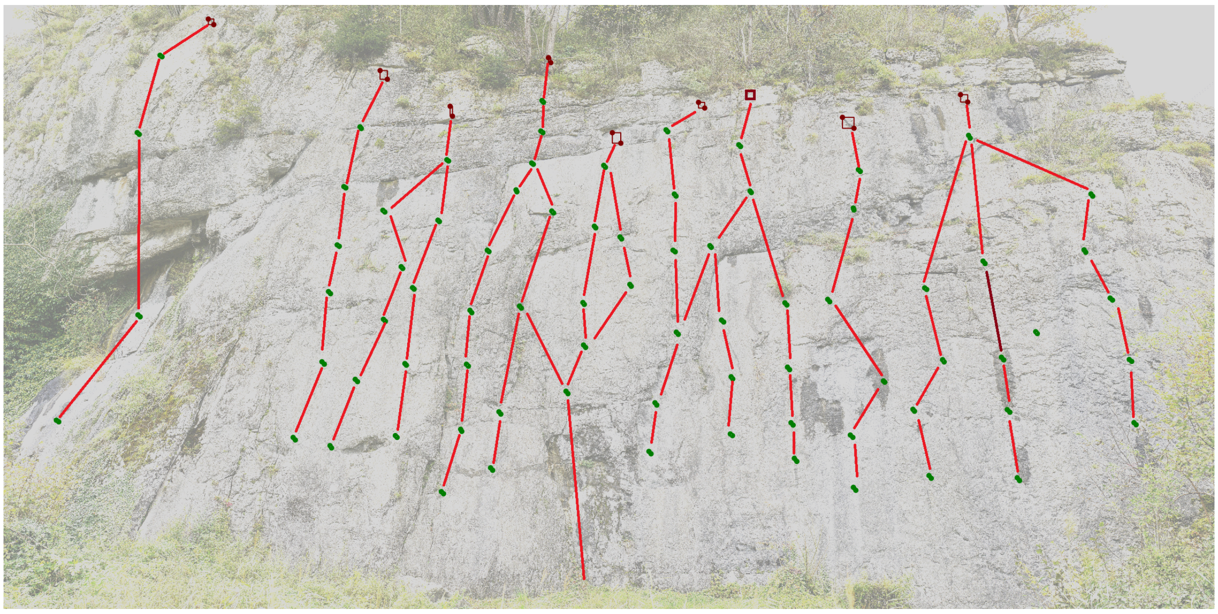


Figure 70: High Brightness Image of the climbing wall with the bolts and anchors detected.

Key challenges include:
- Bolts are not always aligned in a straight line.
- Routes may start and end at different points.
- Routes can merge (converge) or split into multiple paths.

The algorithm is iteratively refined across three approaches: a rule-based method, a Voronoi top-down method, and finally, a robust Voronoi bottom-up solution.

II 5.4.5.1 Rule-Based Algorithm

The algorithm is implemented using the following (Figure 71) steps which are repeated for all routes:

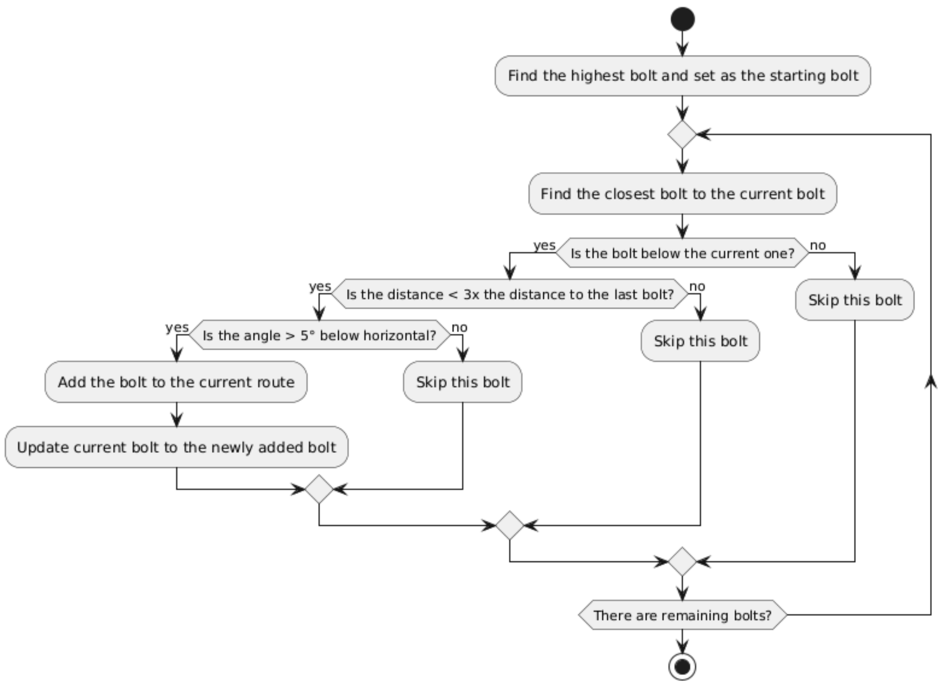

Find the highest bolt and set as the starting bolt
Find the closest bolt to the current bolt
yes
Is the bolt below the current one?
no
yes
Is the distance < 3x the distance to the last bolt?
no
Skip this bolt
yes
Is the angle > 5° below horizontal?
no
Skip this bolt
Add the bolt to the current route
Skip this bolt
Update current bolt to the newly added bolt
There are remaining bolts?

Figure 71: Route detection algorithm flowchart

While this method is straightforward, it struggles with complex scenarios, such as routes merging or splitting. The deterministic nature of the rules limits its ability to handle the variability of real climbing walls.

Moreover, the planning and placement of climbing routes on rock walls do not follow any standardized rules or guidelines. Instead, these placements are often the result of individual experience, intuition, and observation by route setters. As such, the rules used in this algorithm are not official but are derived from practical observations and common patterns in climbing wall designs. This lack of standardization further complicates the algorithm's ability to generalize across different climbing scenarios.

II 5.4.5.2 Voronoi Diagrams

To overcome the limitations of the rule-based approach, the next implementations leverage **Voronoi diagrams** (Figure 72). A Voronoi diagram is a method of partitioning a space into regions based on the proximity to a given set of points.

1. **Definition**: Each point (bolt or anchor) in a Voronoi diagram has a corresponding region consisting of all locations closer to that point than to any other. These regions are called **Voronoi cells**.
2. **Clustering**: By segmenting bolts into Voronoi cells, the algorithm groups bolts based on their spatial relationships, making it easier to define paths and clusters.
3. **Pathfinding**: Voronoi diagrams help establish potential routes by defining connectivity between neighboring cells. These connections are especially useful for managing complex scenarios where routes converge or split.

The Voronoi-based methods simplify clustering and pathfinding, making them more systematic and robust compared to the rule-based approach. However, the success of these methods depends on how the algorithm navigates through the cells and identifies routes.

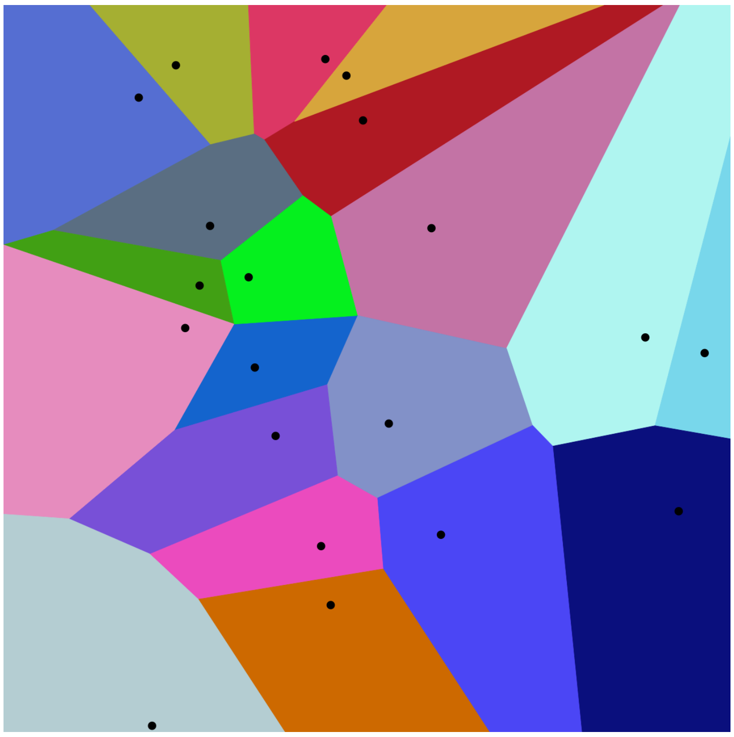


Figure 72: Voronoi Example [30]

II 5.4.5.2.1 Generating a Voronoi Diagram in Code

The Voronoi diagram used in the algorithm is generated with the help of the **voronoijs** library, which provides efficient tools for creating Voronoi diagrams. The following TypeScript in Code-Fragment 24 implementation demonstrates how the library is utilized:

Function to generate a Voronoi diagram using `voronoijs`.

```
import { Voronoi, BoundingBox, Site, Diagram } from 'voronoijs';

export const generateVoronoiDiagram = (sites: Site[], bbox: BoundingBox): Diagram => {
    const voronoi = new Voronoi();
    return voronoi.compute(sites, bbox);
};
```

Code-Fragment 24: Voronoi Diagram Generation Function

The function `generateVoronoiDiagram` computes a Voronoi diagram given a list of `sites` (points) and a `bbox` (bounding box). Below in Code-Fragment 25 is an example of how this function is used in the algorithm:

Processing predictions to generate a Voronoi diagram.

```
const sites: Site[] = predictions.map(prediction => {
    const x = (prediction.x_min + prediction.x_max) / 2;
    const y = (prediction.y_min + prediction.y_max) / 2;
    const id = prediction.id;

    return {x, y, id};
});

const bbox = calculateBBox(job.predictions);
const diagram = generateVoronoiDiagram(sites, bbox);
```

Code-Fragment 25: Voronoi Diagram Processing Code

This approach allows the algorithm to dynamically generate Voronoi cells based on detected bolt positions, ensuring robust route detection.

II 5.4.5.3 Voronoi Bottom-Up Algorithm

The Voronoi Bottom-Up Algorithm is designed to detect climbing routes systematically from the bottom of the climbing wall to the top. This approach ensures better coverage of actual climbing routes, leveraging the observation that routes tend to merge near anchors rather than splitting into new ones. Starting from the bottom provides a higher likelihood of identifying and preserving the structure of multiple routes.

Phase 1: Identifying Starting Cells In the first phase, the algorithm identifies all the starting cells (Figure 73). These are Voronoi cells that touch the bottom boundary of the bounding box (bbox) of the Voronoi diagram. These starting cells serve as the entry points for detecting climbing routes.

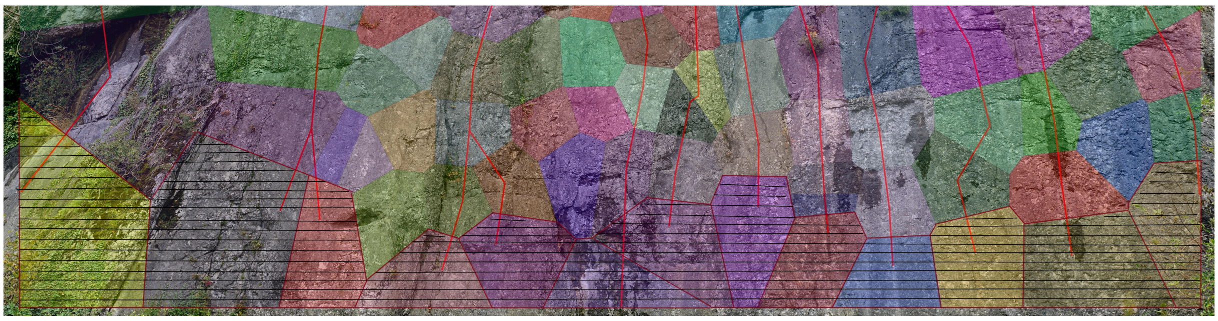


Figure 73: Starting cells

Phase 2: Route Initialization and Queue Processing For each identified starting cell, a new route is initialized. A route contains the following:

- **Route Predictions**: A sequential list of predictions, where each prediction represents a detected bolt or anchor.
- **Position**: The index of the prediction in the route.
- **Metadata**: Additional information about the route, such as whether it is complete.

Each initialized route is added to a queue for processing. Once all starting cells are added to the queue, the algorithm processes the queue iteratively. The process includes:

1. **Retrieve the Current Prediction**: The algorithm dequeues the next route and cell for processing. The cell represents a specific area of the climbing wall, and its associated prediction contains details about a bolt or anchor in that cell.

2. **Add to Route**: The current prediction is added to the route, and its position within the route is updated (Code-Fragment 26).

Adding a prediction to a route with its position.

```
route.routePredictions.push({
    prediction: currentPrediction,
    position: route.routePredictions.length,
});
```

Code-Fragment 26: Route Prediction Addition Code

3. **Handle Whitelist Connections**: Whitelist connections are predefined links between predictions, indicating bolts or anchors that are part of the same climbing path. If the current prediction has connections in the whitelist, the algorithm duplicates the route for each connection and adds these to the queue for further exploration (Code-Fragment 27).

Handling whitelist connections for route predictions.

```
handleWhitelistConnections(
    currentPrediction,
    job,
    diagram,
    predictionCoordinateMap,
    route,
    routeQueue
);
```

Code-Fragment 27: Whitelist Connections Handling Code

4. **Check for Anchor**: If the current prediction is labeled as an anchor, the route is considered complete and added to the list of finalized routes.

5. **Process Neighbors**:
   - Identify neighboring cells above the current prediction to ensure the route progresses upward.
   - Filter out neighbors that are part of the blacklist, which contains connections or predictions that should be excluded from the route.
   - Select the neighboring cell that shares the most horizontal edge with the current cell to maintain a logical climbing path (codeUpwardNeighborsRouting).

Filtering upward neighbors and finding the next cell for routing.

```
const upwardNeighbors = filterUpwardNeighbors(
    currentPrediction,
    getNeighborCells(currentCell, diagram),
    job,
    predictionCoordinateMap,
    existingConnections,
    assignedPredictions
);


const nextCell = findMostHorizontalBolt(currentCell, upwardNeighbors);
if (nextCell) {
    routeQueue.push({ route, currentCell: nextCell });
}
```

Code-Fragment 28: Upward Neighbor Filtering and Routing Logic

6. **Repeat Until Queue is Empty**: The process continues until no routes remain in the queue. At this point, all valid climbing paths starting from the bottom cells have been fully explored.

Phase 3: Processing Unassigned Predictions After generating the routes from the starting cells, some predictions may remain unassigned. To handle these, the algorithm:

1. Selects the lowest unassigned prediction.
2. Adds it to the route queue as a new starting point.
3. Processes the queue again, repeating the steps from phase two.
4. Checks for additional unassigned predictions and repeats the process if necessary.

This ensures that every prediction is evaluated and included in the routes where appropriate. The Figure 74 diagram illustrates the processing of the route queue:

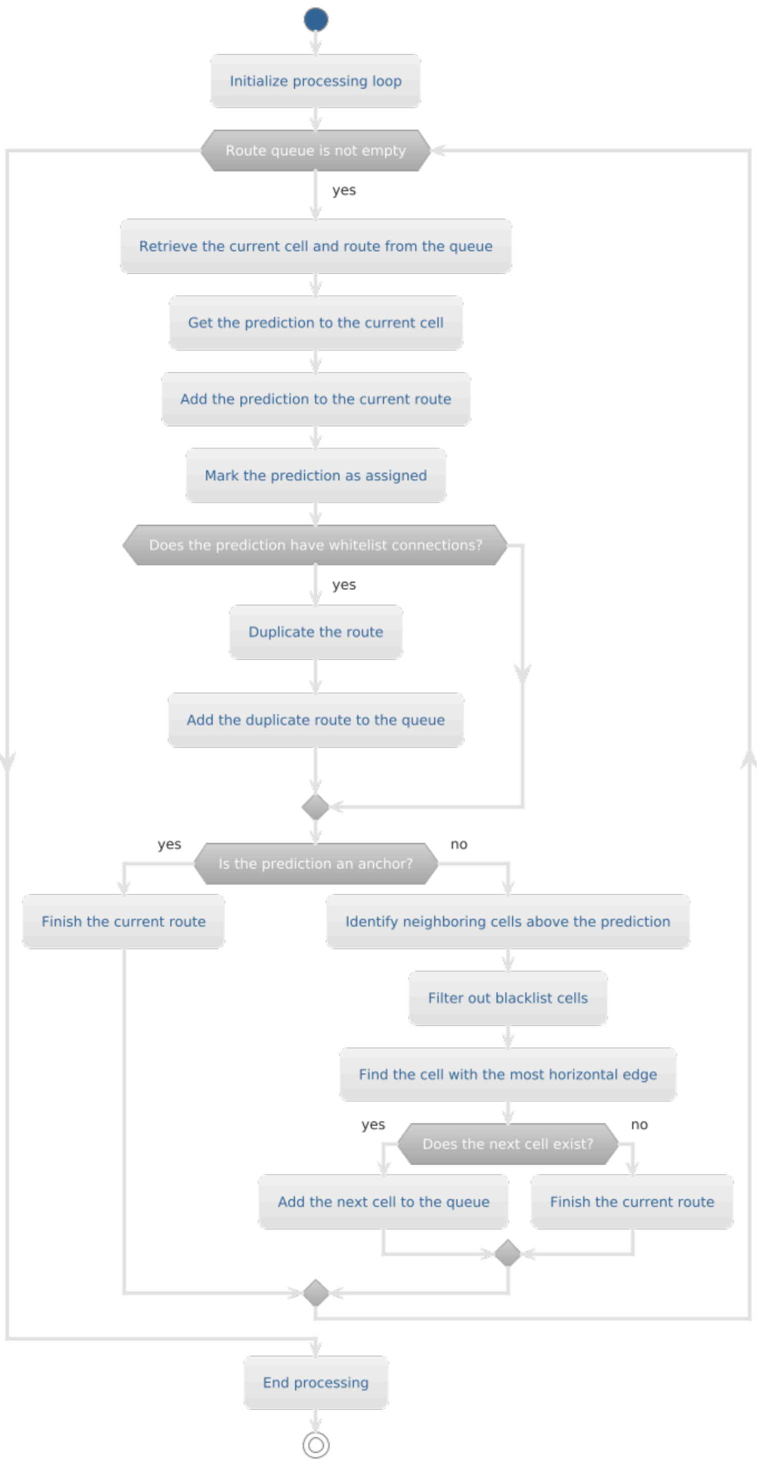

Initialize processing loop
Route queue is not empty
yes
Retrieve the current cell and route from the queue
Get the prediction to the current cell
Add the prediction to the current route
Mark the prediction as assigned
Does the prediction have whitelist connections?
yes
Duplicate the route
Add the duplicate route to the queue
yes
Is the prediction an anchor?
no
Finish the current route
Identify neighboring cells above the prediction
Filter out blacklist cells
Find the cell with the most horizontal edge
yes
Does the next cell exist?
no
Add the next cell to the queue
Finish the current route
End processing

Figure 74: processing the route queue

In a first attempt to detect the routes, the algorithm identifies the starting cells and initializes routes for each one as seen Figure 75:

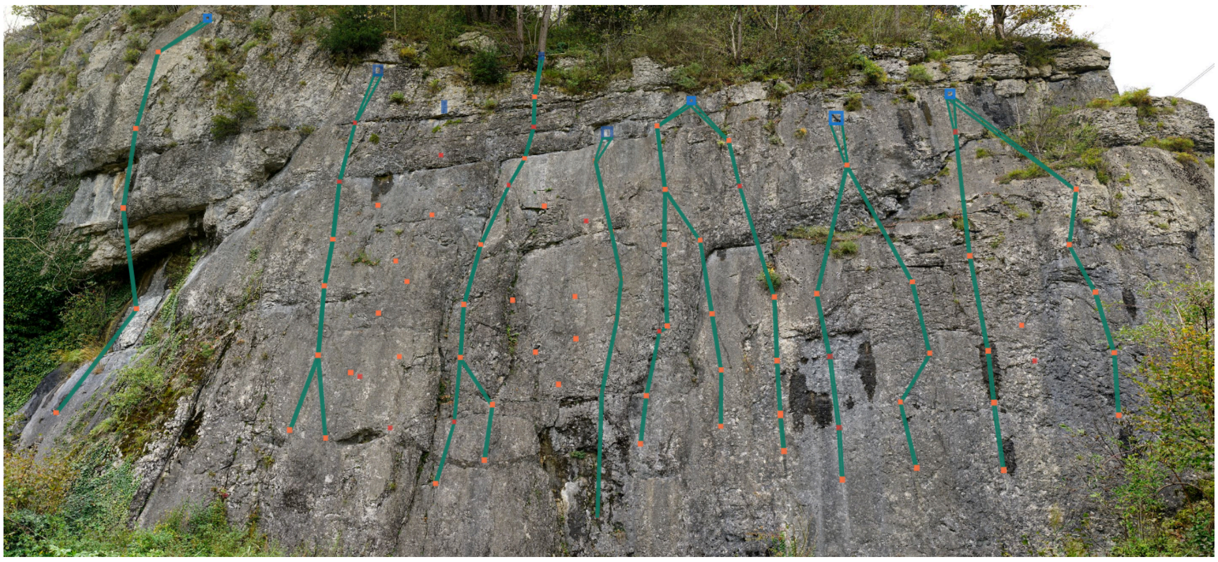


Figure 75: Routes after processing starting cells

As new predictions are processed, the resulting routes incorporate the previously unassigned predictions, leading to comprehensive coverage of the climbing wall, see Figure 76:

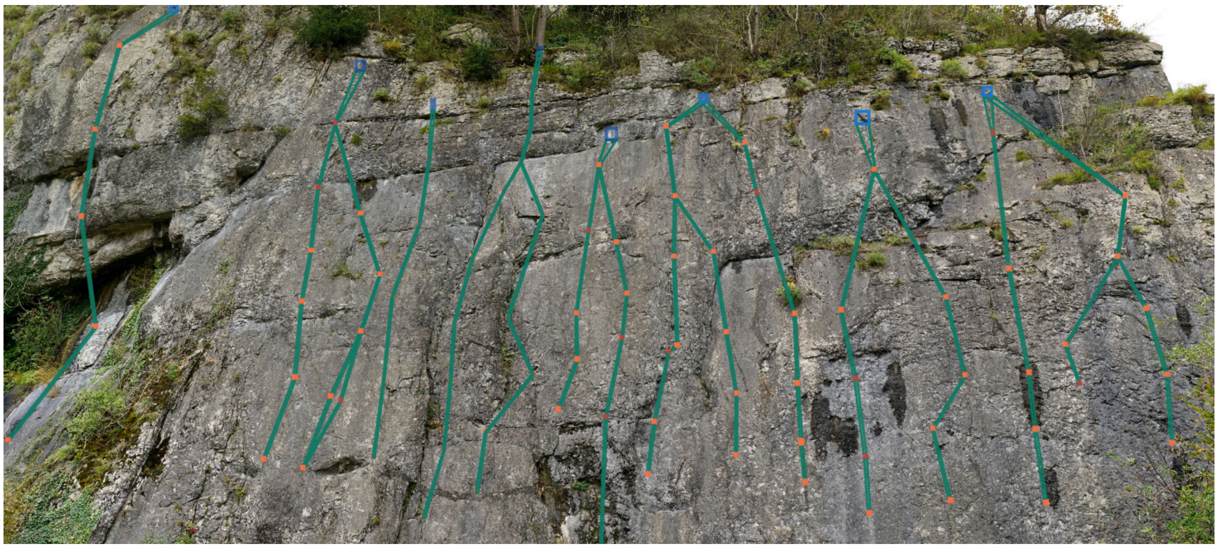


Figure 76: Routes after processing unassigned cells

Why Bottom-Up Instead of Top-Down? The decision to use a bottom-up approach stems from the observation that climbing routes typically merge into fewer anchors at the top. By starting from the bottom, where routes are more numerous and dispersed, the algorithm maximizes the chance of capturing all potential routes. In contrast, a top-down approach risks missing routes that converge early or diverge into sub-paths. The top-down approach is shown in Figure 77.

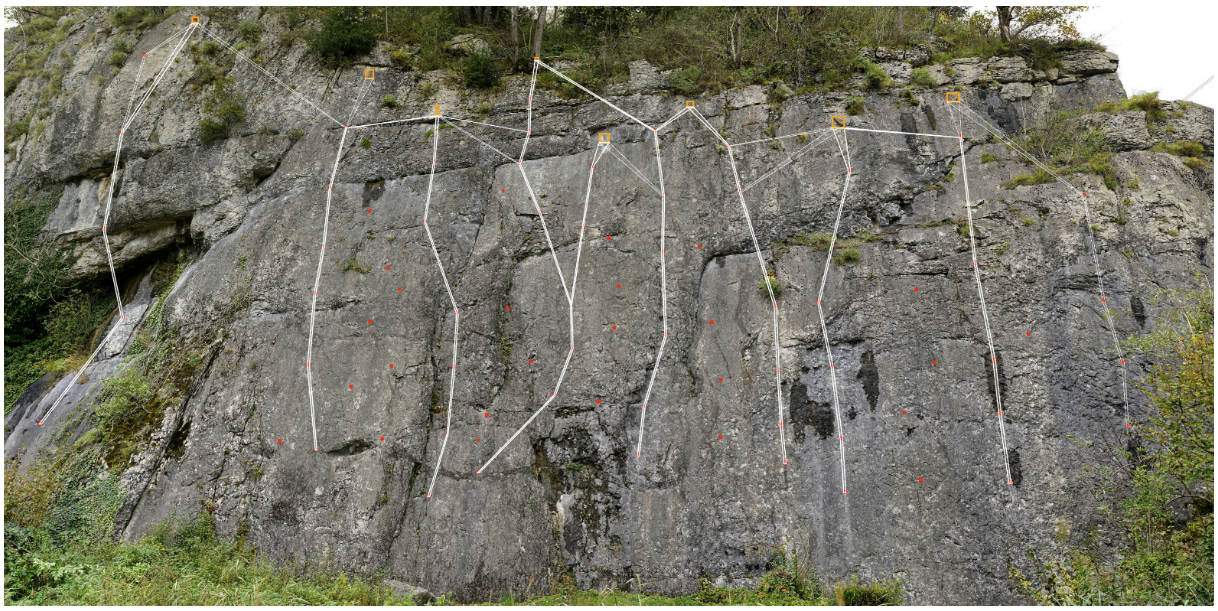


Figure 77: Route detection top to bottom

This bottom-up methodology ensures a thorough and logical mapping of routes, reflecting the natural structure of climbing walls and accommodating real-world complexities like route merges and splits.

II 5.5 Frontend Implementation

The frontend of the BoltFinder project is implemented using the React framework. The frontend is responsible for providing a user-friendly interface for users to interact with the BoltFinder system. The frontend communicates with the backend API to fetch and display data to the user.

II 5.5.1 User flow

The user flow of the BoltFinder frontend is as follows:

1. The user navigates to the BoltFinder website.
2. The user is presented with the home page, where they can upload an image of a climbing wall.
3. The user uploads an image and can navigate to the predict page.
4. The user is presented with the predict page, where they can trigger the prediction job of climbing routes.
5. The user can navigate to the status page to view the status of the prediction job.
6. Once the prediction job is completed, the user can:
   - view and modify the predicted climbing routes on the editor page.
   - view statistics about the prediction job on the statistics page.
   - download a topo image.
   - download a Voronoi image (visualizing the algorithm's decision base).
   - delete the prediction job.

II 5.5.2 Application Structure

The BoltFinder frontend is organized into a modular structure, separating logic, components, and pages for maintainability and scalability. Below is an overview of the main elements:

II 5.5.2.1 Pages

The application consists of the following pages, each responsible for a specific step in the user journey:

1. **Upload Page**: Allows users to upload images of climbing walls. Lists previously uploaded images.
2. **Predict Page**: Lets users trigger a prediction job, configure settings (e.g., model type, confidence threshold), and send data to the backend.
3. **Status Page**: Displays the status of ongoing or completed prediction jobs. Includes error handling for failed jobs.
4. **Edit Page**: Provides an interactive editor to view and modify predicted routes.

II 5.5.2.2 Components

Each page is built from reusable components, which are the building blocks of the application. Here are some key components:

- **Header**: A shared navigation bar visible across all pages.
- **UploadForm**: Handles file uploads on the Upload Page.
- **PredictForm**: Allows the user to select models and confidence thresholds on the Predict Page.
- **StatusTable**: Displays prediction job details on the Status Page.
- **Editor**: A complex component on the Edit Page for interactive route visualization and modification.
- **RoutesTable**: A table component to showcases all the detected routes and interact with them.
- **StatisticsPanel**: Renders statistics for bolts, anchors, routes, etc., used on both the Status and Edit Pages.
- **Notifications**: A global notification system for success/error messages.
- **ErrorBoundary**: Captures and displays error messages for any failed components.

II 5.5.2.3 State Management

The application uses a combination of local component states and global state management tools:

- **React Query**: Fetches, caches, and synchronizes server-side data (e.g., prediction statuses, model configurations).
- **Context API**: Manages global state such as user preferences and currently selected prediction jobs.
- **Axios**: Handles all HTTP requests to the backend API.

II 5.5.3 Configuration

The frontend system for the BoltFinder application relies on a small set of environment variables to define its runtime behavior and API connections. These configurations are essential for ensuring proper communication with the backend and IIIF server. The following environment variables in Code-Fragment 29 are used to configure the frontend application:

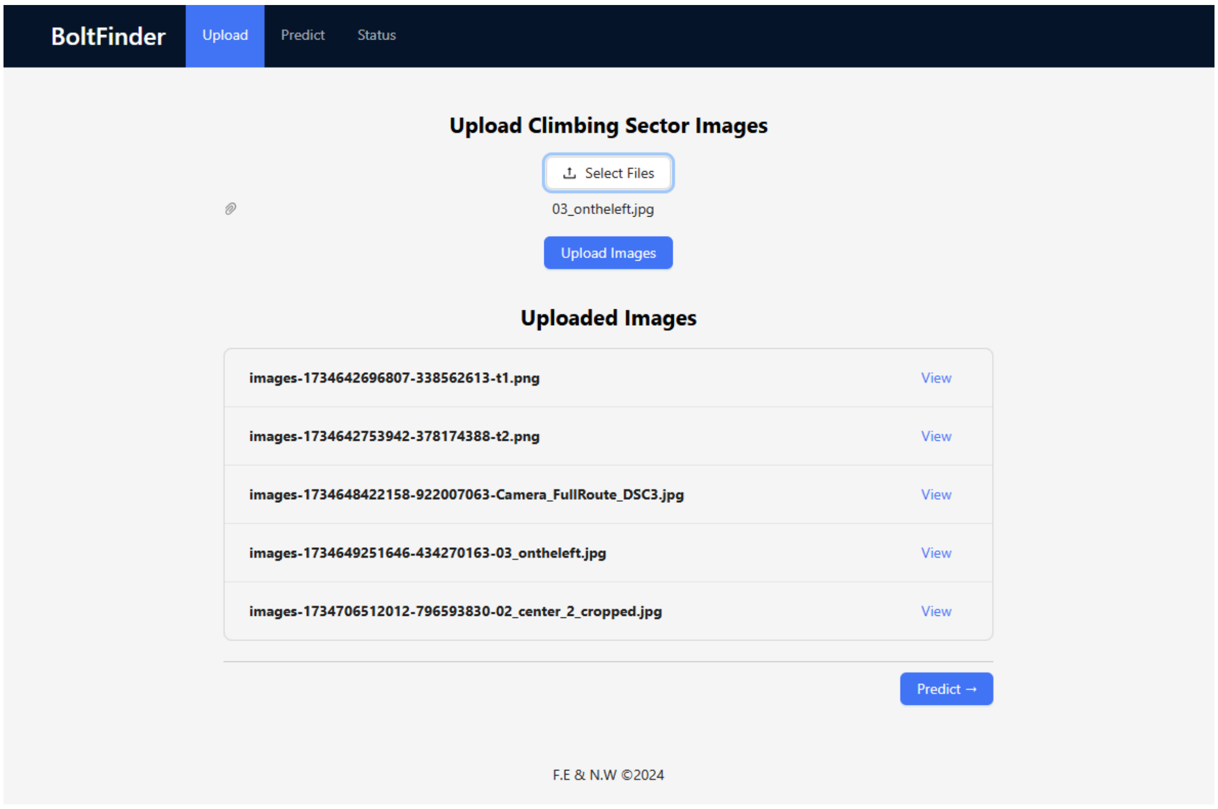
Configuration settings for the BoltFinder frontend.

```
CHOKIDAR_USEPOLLING=true
VITE_BACKEND_API_URL=http://localhost:3000
VITE_IFFF_SERVER_URL=http://localhost:8182
```

Code-Fragment 29: Frontend Configuration

II 5.5.4 Upload Page

The upload page (Figure 78) is the first page the user sees when they navigate to the BoltFinder website. The page allows the user to upload an image of a climbing wall. The previously uploaded images are shown in a list below the upload form, and the images can be viewed from there.



Upload Climbing Sector Images

03_ontheleft.jpg

Uploaded Images

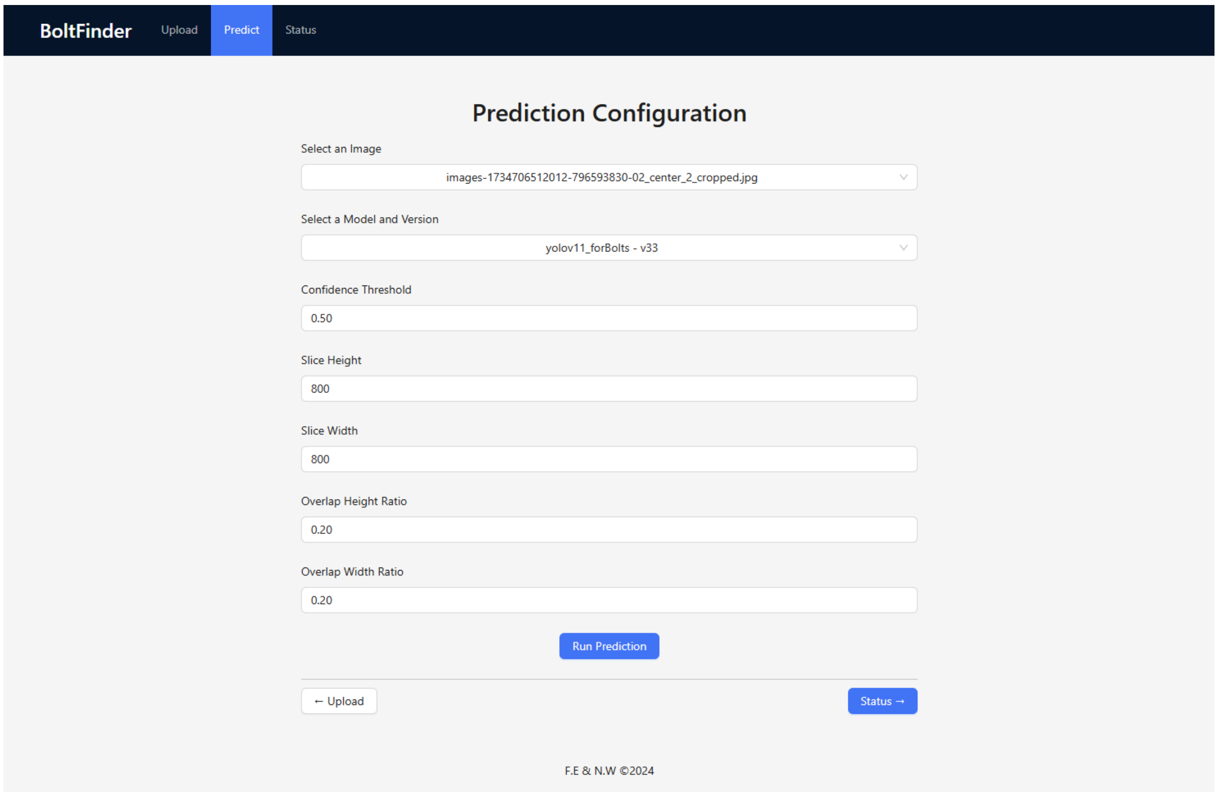
images-1734642696807-338562613-t1.png
images-1734642753942-378174388-t2.png
images-1734648422158-922007063-Camera_FullRoute_DSC3.jpg
images-1734649251646-434270163-03_ontheleft.jpg
images-1734706512012-796593830-02_center_2_cropped.jpg

F.E & N.W ©2024

Figure 78: Home / Upload Page

II 5.5.5 Predict Page

The predict page (Figure 79) is where the user can trigger the prediction job of the climbing routes. The user can select the prediction model to use and set the confidence threshold for the predictions.



Prediction Configuration

Select an Image: images-1734706512012-796593830-02_center_2_cropped.jpg

Select a Model and Version: yolov11_forBolts - v33

Confidence Threshold: 0.50

Slice Height: 800

Slice Width: 800

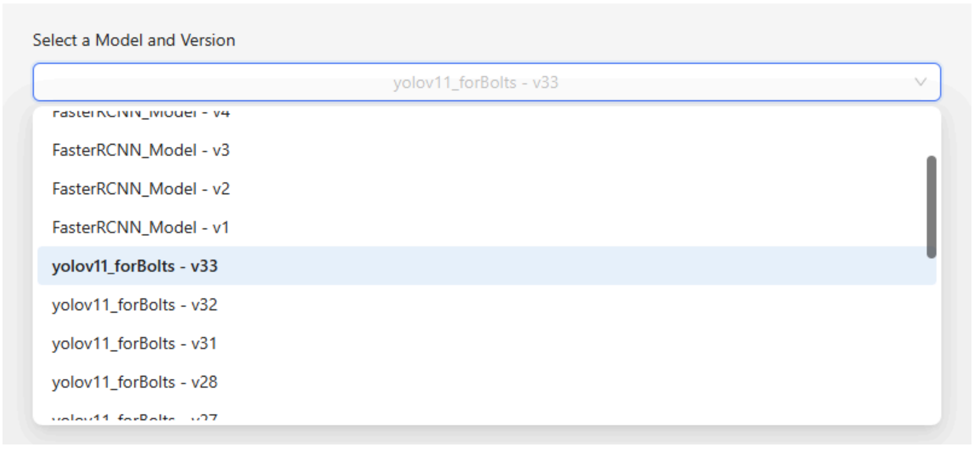
Overlap Height Ratio: 0.20

Overlap Width Ratio: 0.20

F.E & N.W ©2024

Figure 79: Predict Page

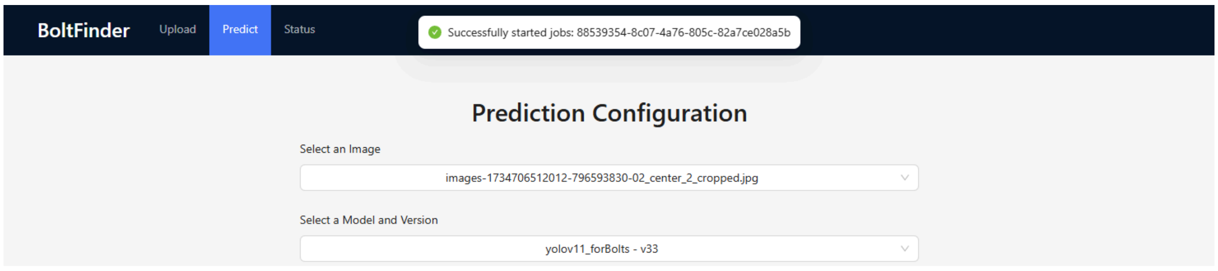
The model version selection dropdown (Figure 80) allows the user to select the Faster R-CNN or YOLOv11 model version to use for the prediction job. The dropdown is shown in Figure 80.



Select a Model and Version: yolov11_forBolts - v33

Figure 80: Predict Page Drop Down

Throughout the BoltFinder frontend, there are notification messages that inform the user about the status of their actions. For example, when the user triggers the prediction job, a message is displayed to inform the user that the job has been successfully started. The message is shown in Figure 81.



Successfully started jobs: 88539354-8c07-4a76-805c-82a7ce028a5b

Prediction Configuration

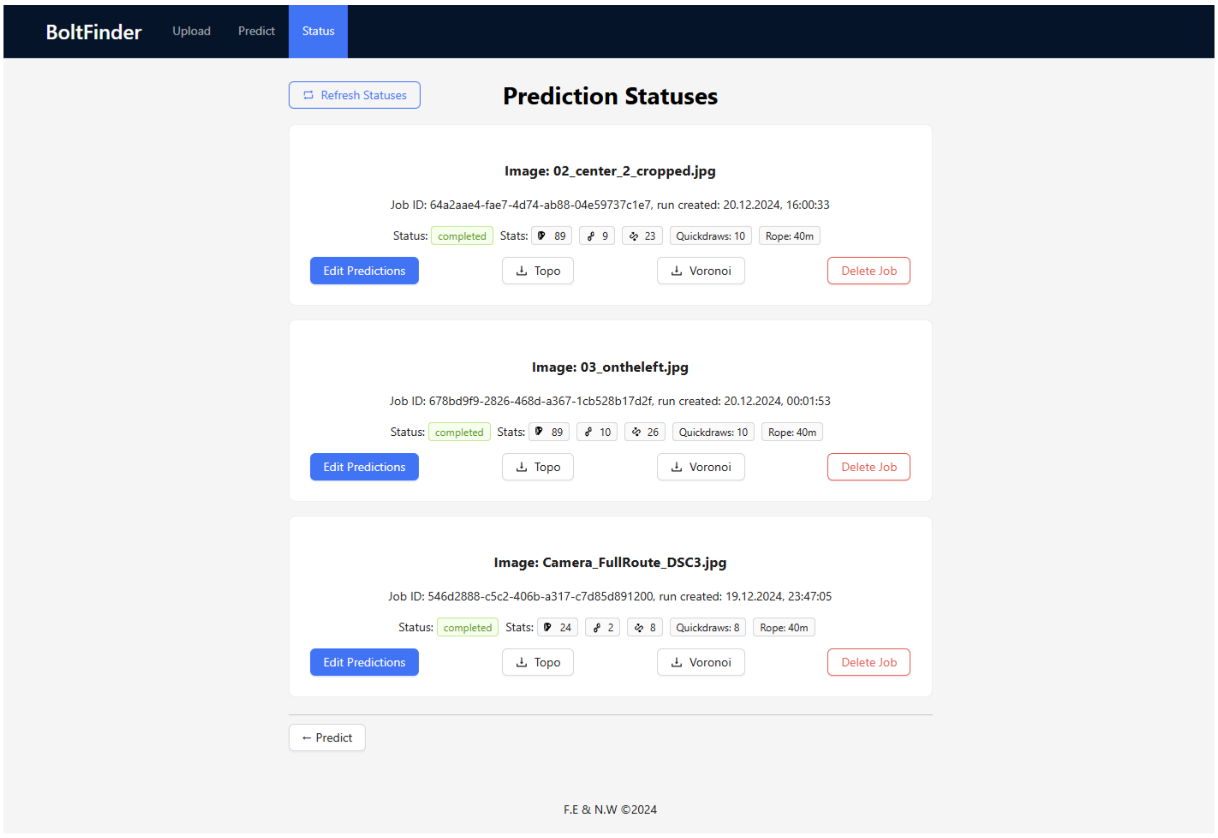
Select an Image: images-1734706512012-796593830-02_center_2_cropped.jpg

Select a Model and Version: yolov11_forBolts - v33

Figure 81: Predict Job Success Message

II 5.5.6 Status Page

The status page (Figure 82) shows the user the status of the prediction jobs.


BoltFinder Upload Predict Status

Refresh Statuses

Prediction Statuses

Image: 02_center_2_cropped.jpg

Job ID: 64a2aae4-fae7-4d74-ab88-04e59737c1e7, run created: 20.12.2024, 16:00:33

Status: completed Stats: 89 9 23 Quickdraws: 10 Rope: 40m

Edit Predictions Topo Voronoi Delete Job

Image: 03_ontheleft.jpg

Job ID: 678bd9f9-2826-468d-a367-1cb528b17d2f, run created: 20.12.2024, 00:01:53

Status: completed Stats: 89 10 26 Quickdraws: 10 Rope: 40m

Edit Predictions Topo Voronoi Delete Job

Image: Camera_FullRoute_DSC3.jpg

Job ID: 546d2888-c5c2-406b-a317-c7d85d891200, run created: 19.12.2024, 23:47:05

Status: completed Stats: 24 2 8 Quickdraws: 8 Rope: 40m

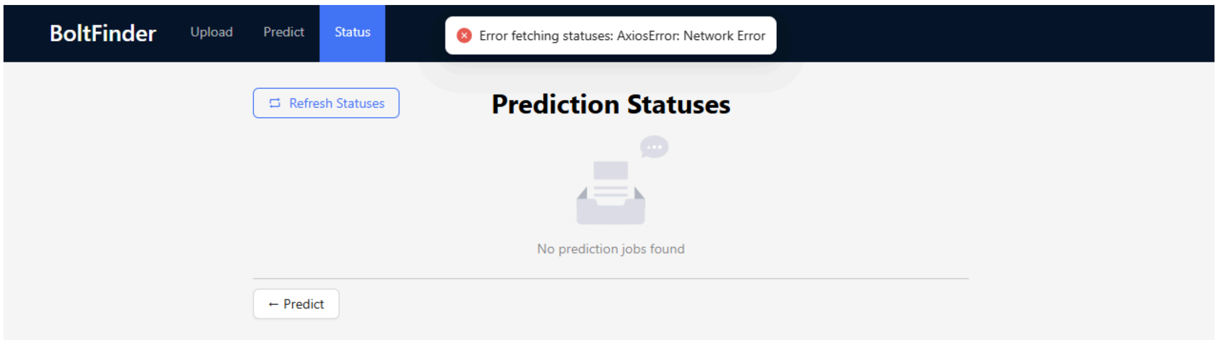
Edit Predictions Topo Voronoi Delete Job

← Predict

F.E & N.W ©2024

Figure 82: Status Page

In the case of network issues or empty responses from the backend, the user is informed about the error and an empty placeholder component is shown. The error message is shown in Figure 83.


BoltFinder Upload Predict Status

Error fetching statuses: AxiosError: Network Error

Refresh Statuses

Prediction Statuses

No prediction jobs found

← Predict

Figure 83: Showing Errors

The status page as well as the edit page feature a stats component (Figure 84) which shows the user the number of bolts, anchors, routes and additionally the maximum amount of needed quickdraws and the rope length needed for the longest route. The quickdraw amount is calculated by the amount of bolts and anchors in one route. The rope length is calculated by the amount of quickdraws and the distance between the bolts. The calculation is shown in Code-Fragment 30.

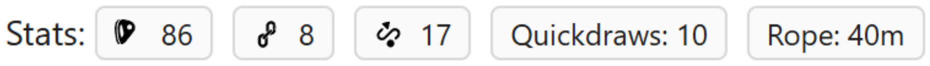


Figure 84: Statistics

Rope length calculation for the longest route.

```
const determineRopeLength = (quickdrawCount: number): RopeLength => {
    const RouteLength = 2 * (1 + quickdrawCount * 1.5);
   // 1m to the first bolt, 1.5m between each bolt, times two for the return trip
    if (RouteLength >= 80) return 80;
    if (RouteLength >= 70) return 70;
    if (RouteLength >= 60) return 60;
    return 40;
};
```

Code-Fragment 30: Rope Length Calculation

II 5.5.6.0.1 Topology Image Example

Over the Topo download button as seen in Figure 82 the user can download a topology Image. This image shows the climbing wall with the predicted climbing routes. The image is shown in Figure 85.

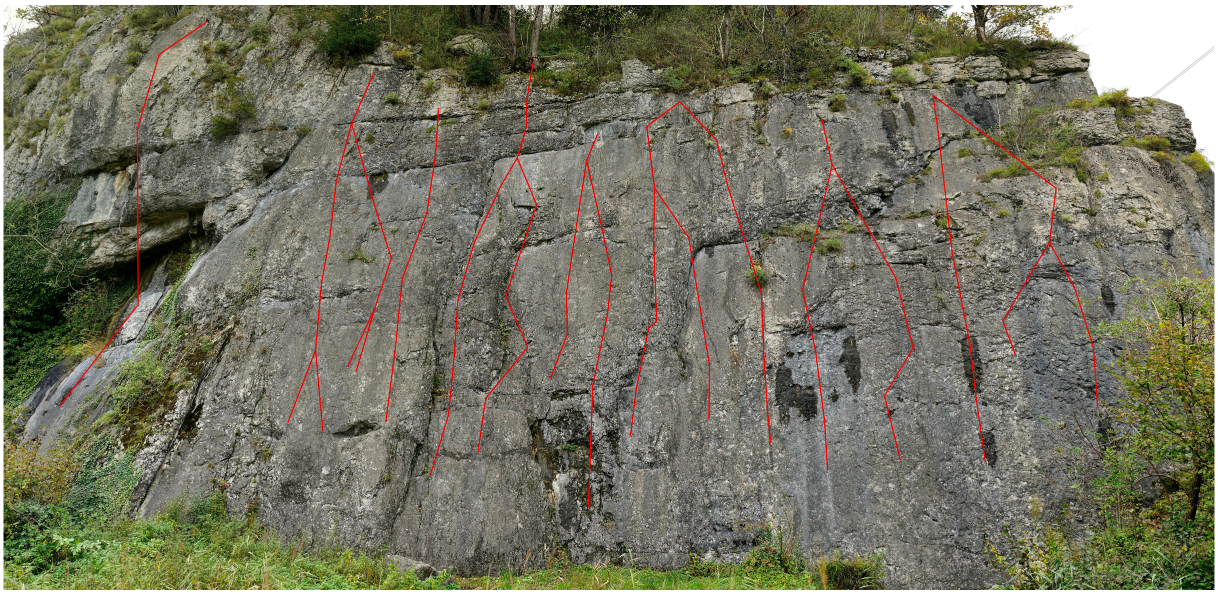


Figure 85: Example of a Topology Image showing predicted climbing routes.

II 5.5.6.1 Additional Features

The status page includes the following additional functionalities:

1. **Job Deletion**:
   - Users can delete a prediction job if it is no longer needed.
   - This action is irreversible, and the user is prompted for confirmation before the deletion is performed.
2. **Image Downloads**:
   - The user can download:
     - A **Voronoi image**: A diagram visualizing the decision-making process of the algorithm.

II 5.5.6.1.1 Voronoi Image Example

The Voronoi image download button is shown in Figure 82. The Voronoi image visualizes the algorithm’s decision-making process by highlighting the areas of influence for each prediction. The image is shown in Figure 86.

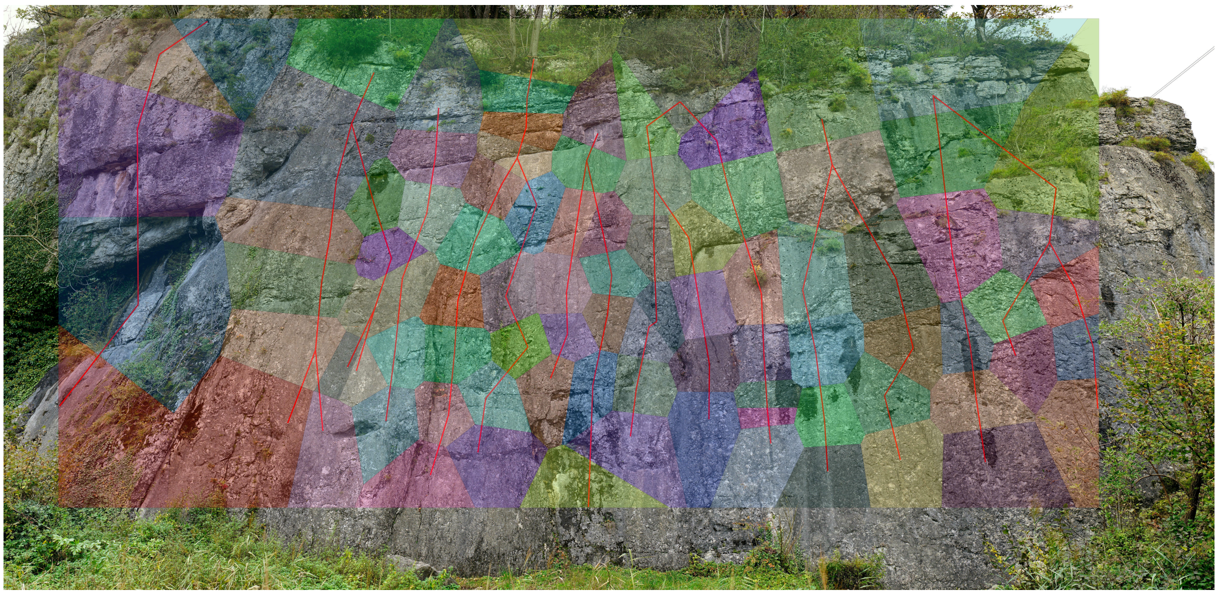
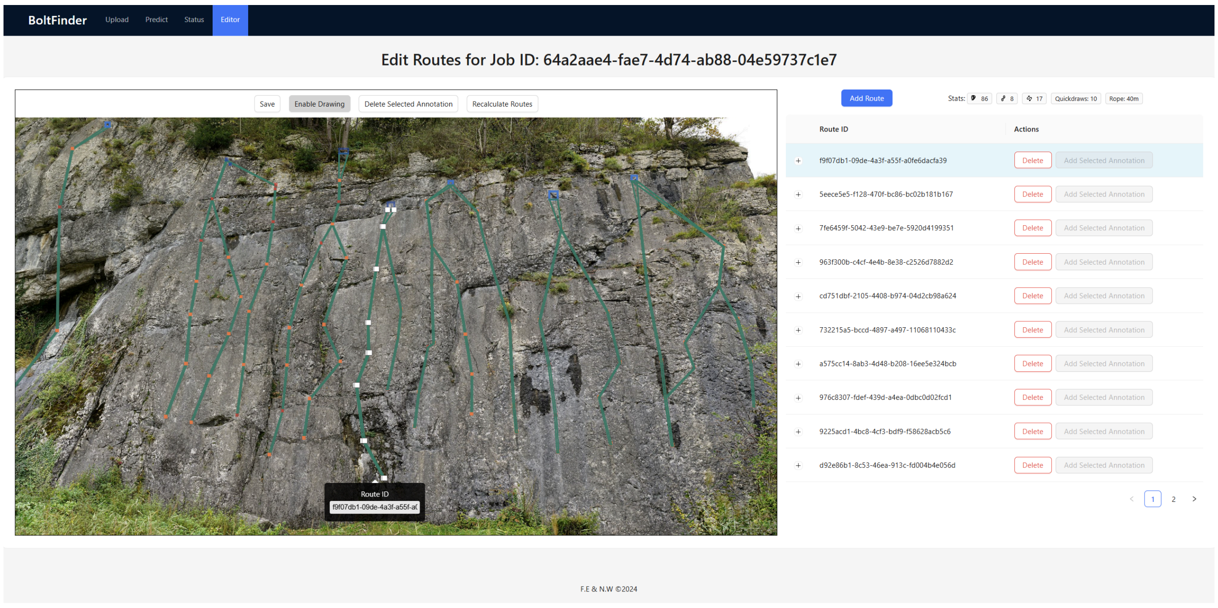


Figure 86: Example of a Voronoi Image visualizing the algorithm's decision-making.

II 5.5.7 Edit Page

The Edit Page (Figure 87) features an advanced image editing experience using several specialized libraries and services to handle climbing wall images and annotations efficiently.



Edit Routes for Job ID: 64a2aae4-fae7-4d74-ab88-04e59737c1e7

	Route ID	Actions
+	f9f07db1-09de-4a3f-a55f-a0fe6dacfa39	Delete Add Selected Annotation
+	5eece5e5-f128-470f-bc86-bc02b181b167	Delete Add Selected Annotation
+	7fe6459f-5042-43e9-be7e-5920d4199351	Delete Add Selected Annotation
+	963f300b-c4cf-4e4b-8e38-c2526d7882d2	Delete Add Selected Annotation
+	cd751dbf-2105-4408-b974-04d2cb98a624	Delete Add Selected Annotation
+	732215a5-bccd-4897-a497-11068110433c	Delete Add Selected Annotation
+	a575cc14-8ab3-4d48-b208-16ee5e324bcb	Delete Add Selected Annotation
+	976c8307-fdef-439d-a4ea-0dbc0d02fcd1	Delete Add Selected Annotation
+	9225acd1-4bc8-4cf3-bdf9-f58628acb5c6	Delete Add Selected Annotation
+	d92e86b1-8c53-46ea-913c-fd004b4e056d	Delete Add Selected Annotation

Figure 87: Edit Page

The page integrates the following technologies:

- **OpenSeadragon**: Provides deep zoom capabilities for navigating high-resolution climbing wall images. This ensures users can smoothly zoom and pan the image while maintaining clarity for fine-grained adjustments.
- **Annotorious**: Built on top of OpenSeadragon, Annotorious is used for adding and editing annotations such as routes, bolts, and anchors directly on the climbing wall image.
- **Cantaloupe Image Server**: Serves high-resolution images via the IIIF (International Image Interoperability Framework) protocol, ensuring optimal image delivery and compatibility with OpenSeadragon.
- **IIIF Protocol**: Enables dynamic image tiling and streaming for efficient rendering of large climbing wall images.

II 5.5.7.1 How Image Tiling Works:

The IIIF protocol and Cantaloupe server divide the image into tiles of varying resolutions, known as **tile levels**. The **tile level** loaded depends on the zoom level of the image. This technique is also used in Google Maps [31] to be able to zoom in and out of the map without loading the entire planet in detail. The Figure 88 illustrates a simplified representation of the various image resolutions that the Cantaloupe server stores and provides based on the uploaded image.

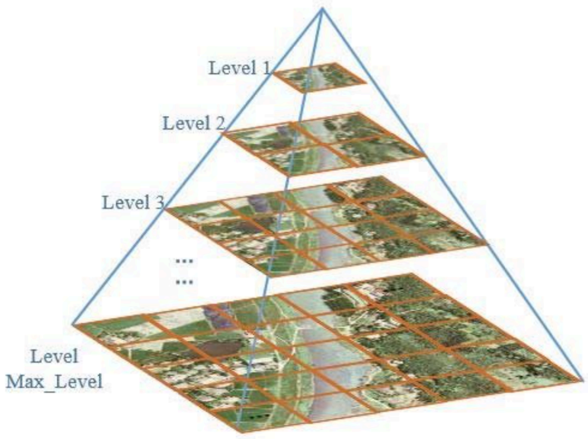


Figure 88: Sketch Map of Tile-Pyramid Structure [32]

The goal is to reduce memory usage and optimize performance by loading only the visible portions of the image, rather than the entire image at high resolution. The approach is also comparable to mipmap loading in 3D graphics, where textures are loaded at varying resolutions based on the distance from the camera [33].

1. **Zoom-Level-Based Tile Loading**:

- At lower zoom levels, larger tiles are loaded, covering broader regions of the climbing wall image.
- At higher zoom levels, smaller tiles are loaded, providing more detailed sections of the visible area.

2. **Dynamic Loading**:

- Only the tiles corresponding to the visible portion of the image are loaded into the client, ensuring efficient rendering.
- As the user pans or zooms further, additional tiles are fetched dynamically from the Cantaloupe server.

The following Figure 89 illustrate the tile-loading process at different zoom levels:

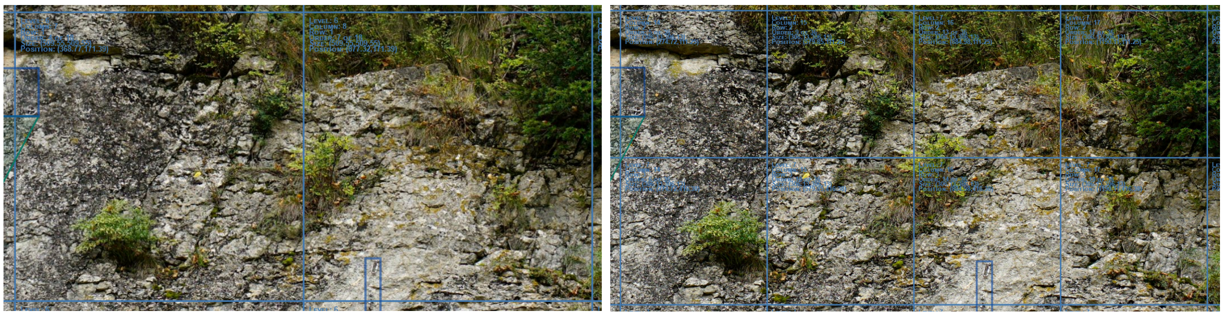


Figure 89: "On the left, the zoom level is lower (Level 6), so larger tiles are loaded. On the right, the zoom level is higher (Level 7), so smaller tiles are loaded. Only the visible portion of the image is affected."

II 5.5.7.2 Deep Zoom and Annotation

1. **Image Handling with Cantaloupe and IIIF**:

- The high-resolution climbing wall images are hosted on a Cantaloupe image server.
- Using the IIIF protocol, images are delivered in tiles, allowing OpenSeadragon to load only the visible portions of the image as users zoom and pan.

2. **Annotation Management**:

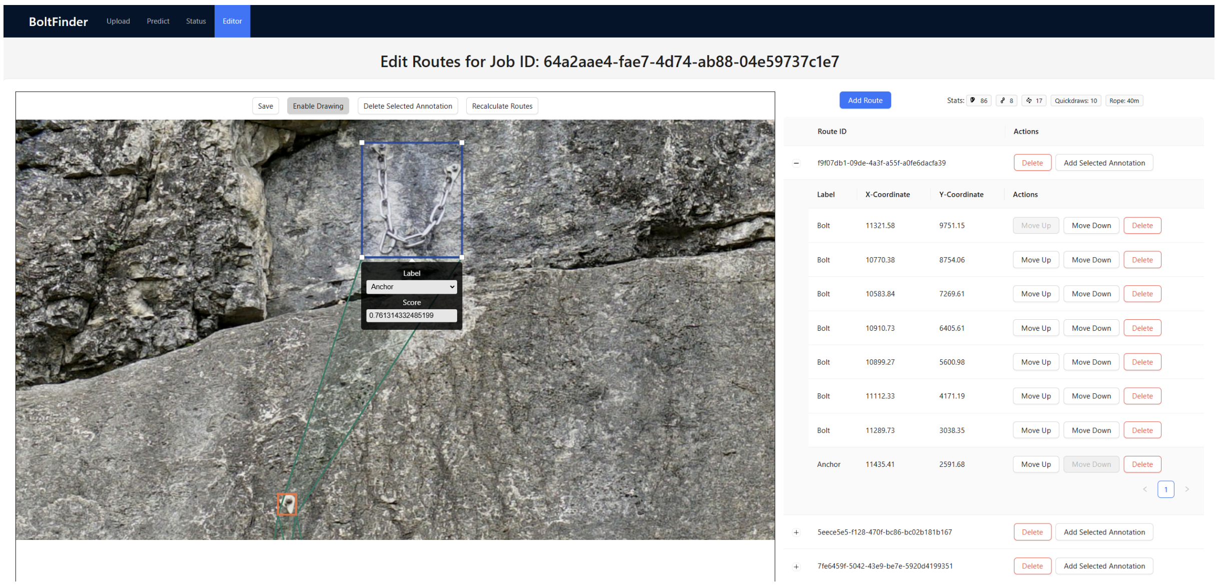
- Annotorious, integrated with OpenSeadragon, overlays an annotation layer on the climbing wall image.
- Users can highlight routes, bolts, and anchors or modify predictions interactively. Annotation styling dynamically reflects prediction confidence (e.g., red for low confidence, orange for high confidence).

II 5.5.7.3 Features Enabled by These Libraries:

- **Dynamic Loading**: Images are delivered in real-time as users zoom and pan across the climbing wall.
- **Annotation Management**: Users can create, edit, or delete annotations for routes and bolts.
- **Performance Optimization**: By leveraging IIIF and tile-based image streaming, the application handles large images efficiently without overloading the client.

II 5.5.7.4 Further Edit Page Features

When unfolding a route, the individual bolts and anchors inside that route are displayed. Hovering over a item selects it in the editor like the anchor in Figure 90.



Edit Routes for Job ID: 64a2aae4-fae7-4d74-ab88-04e59737c1e7

Score: 0.761314332485199

Stats: 86 | 8 | 17 | Quickdraws: 10 | Rope: 40m

Route ID	Actions
f9f07db1-09de-4a3f-a55f-a0fe6dacfa39	Delete, Add Selected Annotation

Label	X-Coordinate	Y-Coordinate	Actions
Bolt	11321.58	9751.15	Move Up, Move Down, Delete
Bolt	10770.38	8754.06	Move Up, Move Down, Delete
Bolt	10583.84	7269.61	Move Up, Move Down, Delete
Bolt	10910.73	6405.61	Move Up, Move Down, Delete
Bolt	10899.27	5600.98	Move Up, Move Down, Delete
Bolt	11112.33	4171.19	Move Up, Move Down, Delete
Bolt	11289.73	3038.35	Move Up, Move Down, Delete
Anchor	11435.41	2591.68	Move Up, Move Down, Delete

Route ID	Actions
5eece5e5-f128-470f-bc86-bc02b181b167	Delete, Add Selected Annotation
7fe6459f-5042-43e9-be7e-5920d4199351	Delete, Add Selected Annotation

Figure 90: Editor Predictions inside Route

As shown in Figure 91 the Editor shows lower confidence predictions in red and higher confidence predictions in orange.

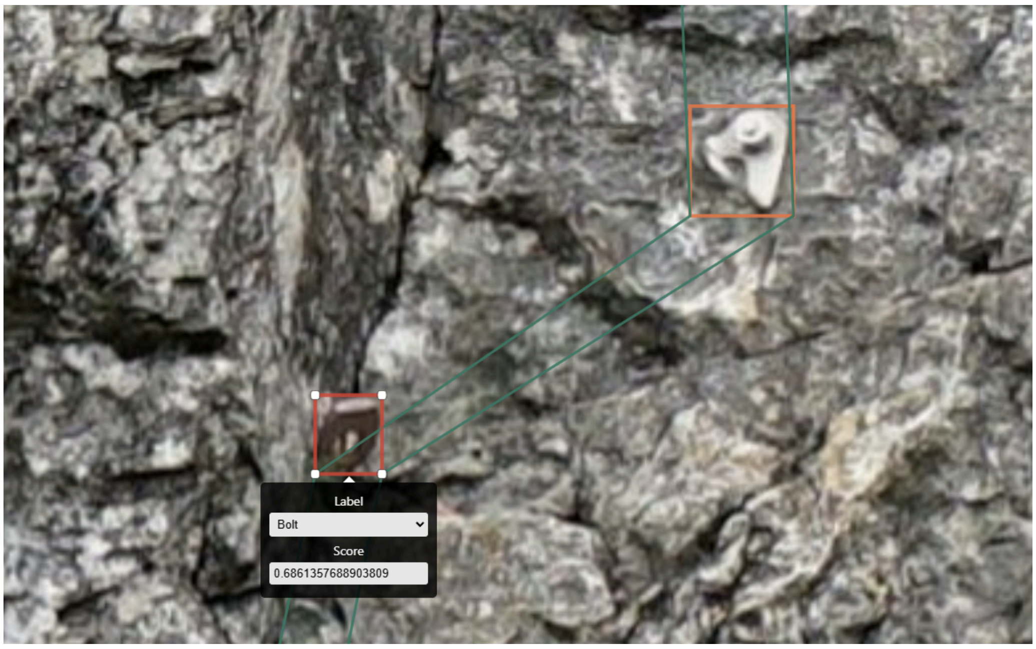
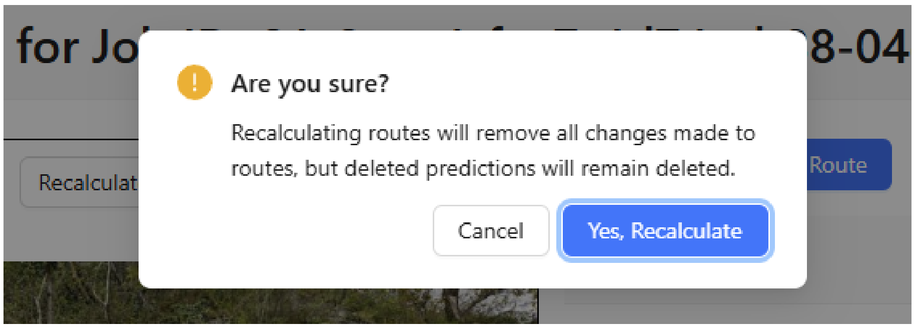


Figure 91: Lower Threshold Red

The Editor page also features a recalculate routes button. This button is used to recalculate the routes based on the current predictions and delete any previously set routes. Since this action is irreversible, a confirmation dialog is shown to the user before recalculating the routes. The confirmation dialog is shown in Figure 92.


Are you sure?

Recalculating routes will remove all changes made to routes, but deleted predictions will remain deleted.

Figure 92: Recalculate Routes

Also for the Editor page there are notification messages if user actions are successful or not. For example, if the user successfully recalculates the routes, a success message is displayed like shown in Figure 93.

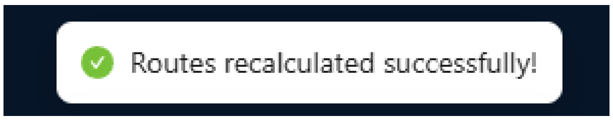

Routes recalculated successfully!

Figure 93: Success Recalculation

II 5.6 Infrastructure

The software components can be run manually, but it is recommended to use the provided docker-compose file. The docker-compose setup starts the backend and frontend services and connects them via a Docker network.

The following diagram (Figure 94) shows the infrastructure during the development of the project. There are four hosts involved:

- **gitlab.ost.ch**: The GitLab server that hosts the repositories and serves as the Docker image registry.
- **mlflow-ba.infs.ch**: The MLFlow server that tracks and manages machine learning experiments.
- **hn601948.ost.ch**: The Windows machine used for training runs and hosting the production Docker Compose setup.
- **localhost**: The development machine used during implementation and testing.

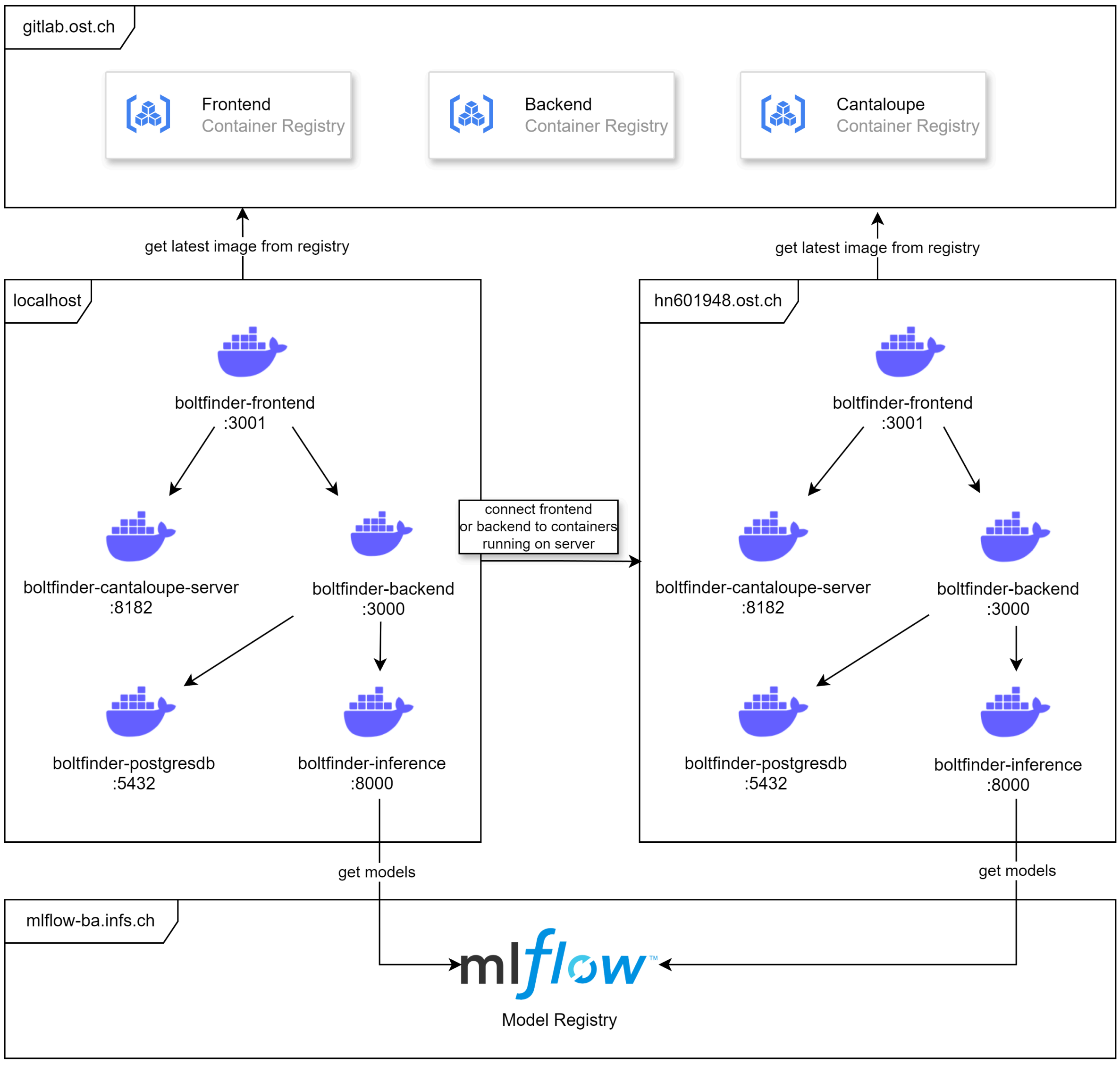


Figure 94: Docker infrastructure

II 5.6.1 Implementation Overview

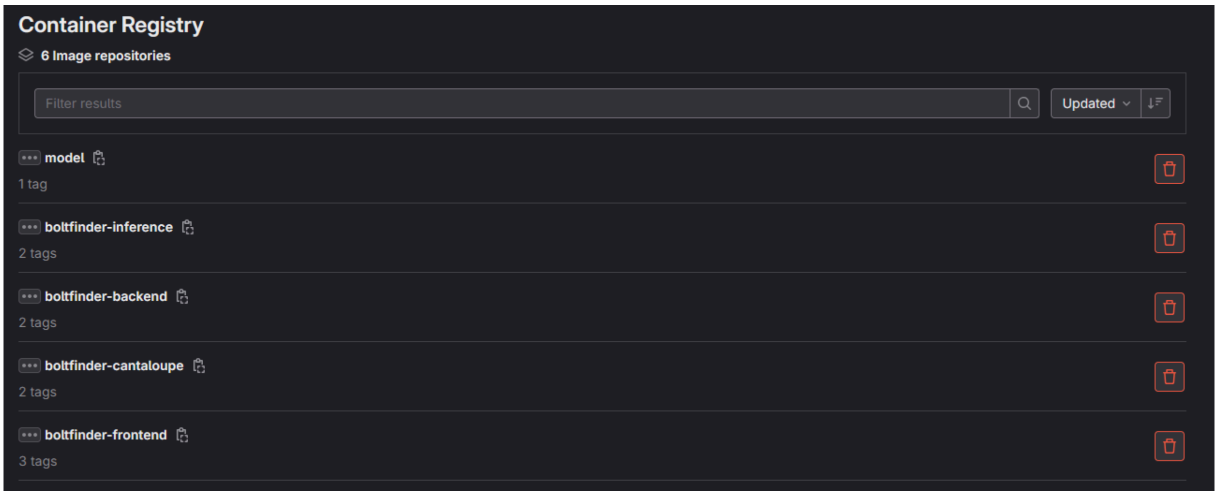
The implementation process involves setting up and managing containers for various BoltFinder components, ensuring seamless integration and deployment.

II 5.6.2 Key Components

- **Frontend**: Implements the user interface using React, deployed via Docker.
- **Backend**: Provides API logic using Express.js for serving the application data.
- **Inference Service**: Runs the ML models for bolt detection, leveraging GPU resources.
- **Database**: Stores the application's persistent data using PostgreSQL.
- **Image Server**: Serves processed images via the Cantaloupe server.

II 5.6.3 Image Repository

All Docker images are stored and managed in the GitLab image registry (Figure 95). These images are tagged and pulled as part of the Docker Compose setup.


Container Registry

6 Image repositories

model
1 tag

boltfinder-inference
2 tags

boltfinder-backend
2 tags

boltfinder-cantaloupe
2 tags

boltfinder-frontend
3 tags

Figure 95: Docker image repository

II 5.6.4 Summary

The implementation ensures that all components are containerized and operate efficiently within the Docker infrastructure, simplifying both development and production deployment.

II 5.7 Test Concept

Quality assurance is an essential part of software development. Its aim is to ensure that the developed software meets the specified requirements and functions reliably, efficiently, and error-free.

This chapter highlights the quality assurance measures that are carried out as part of this work. The **focus** regarding testing for this POC lies on the route algorithm and the evaluation of the model predictions. Testing concentrates on three core components of the system: the **inference API**, the **backend API**, and the **frontend**, as these form the foundation of the application.

II 5.7.1 Overview of Testing

Testing for this proof of concept is carried out using a combination of **manual tests** and **automated tests** to validate the critical components. The testing approach is summarized as follows:

Manual Tests:

- Developers manually test the system end-to-end to verify functionality across all components.
- Manual tests validate the implementation of use cases (Chapter II 2.3 Figure 17) and functional requirements (Chapter II 2.5).
- The frontend, backend, and inference API are tested as part of these manual flows, ensuring smooth communication and accurate results.

Component-Specific Testing:

- For each critical component—**inference API**, **backend API**, and **frontend**—dedicated testing strategies are employed:
  - **Inference API**: Testing focuses on validating model predictions, input/output structures, and performance under edge cases.
  - **Backend API**: Testing ensures proper communication with the inference API, database, and frontend, with a focus on data integrity and error handling. The framework in place for testing is Jest.
  - **Frontend**: Testing concentrates on user interactions, integration with the backend, and responsiveness of the UI.

Integration Testing:

- Integration testing is carried out to validate that the system components worked together as expected. Key integration points include:
- Uploading climbing wall images via the frontend and ensuring they are correctly processed by the backend and inference API.
- Retrieving prediction results from the inference API and displaying them accurately on the frontend.
- Propagating errors between components (e.g., inference API to backend, backend to frontend) to ensure meaningful error messages.

Automated Testing:

- Automated unit tests are written for the **backend API** and **inference API** to validate endpoints, data flow, and edge cases. These tests form the foundation for regression testing in future iterations.
- Due to the POC nature of the project, automated end-user tests (e.g., Playwright or Cypress) are not implemented for the frontend, but manual tests are carried out for key workflows.

The following sections provide detailed testing strategies for each critical component of the system:

II 5.8 Inference API Testing

The **Inference API** is a core component of the BoltFinder system, responsible for running object detection models to predict climbing routes, bolts, and anchors. It serves as the entry point for model inference and ensures that predictions are accurate and reliable. Given its importance, the inference API undergoes extensive testing to validate its behavior across various scenarios.

II 5.8.1 Overview of Testing

The following aspects of the inference API are tested:

1. **Endpoint Testing**:

- Verifies that all API endpoints (e.g., `/models`, `/predict`) function correctly and return expected results.
- Ensures the API handles both valid requests and edge cases, such as invalid inputs or exceptions in the model service.

2. **Inference Service Testing**:

- Tests the core logic for managing models, including loading and caching models.
- Validates the behavior of inference engines (Faster R-CNN and YOLOv11) when running predictions.

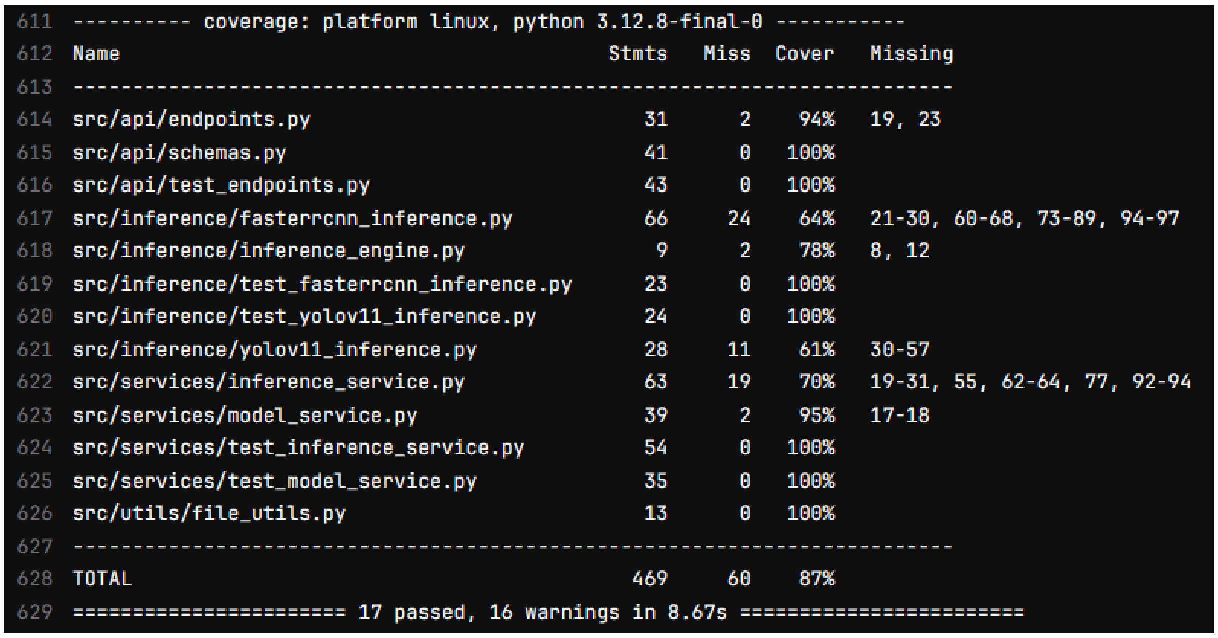
3. **Integration with Backend**:

- Verifies that the backend interacts correctly with the inference API and passes predictions back to the frontend.

4. **Performance Testing**:

- Ensures the inference API responds within acceptable time limits for various input sizes and configurations.

The screenshot below (Figure 96) shows the **test coverage** for the inference API, confirming that all critical paths in the code are covered:



```
---------- coverage: platform linux, python 3.12.8-final-0 -----------
Name                                     Stmts   Miss  Cover   Missing
----------------------------------------------------------------------
src/api/endpoints.py                        31      2    94%   19, 23
src/api/schemas.py                          41      0   100%
src/api/test_endpoints.py                   43      0   100%
src/inference/fasterrcnn_inference.py       66     24    64%   21-30, 60-68, 73-89, 94-97
src/inference/inference_engine.py            9      2    78%   8, 12
src/inference/test_fasterrcnn_inference.py  23      0   100%
src/inference/test_yolov11_inference.py     24      0   100%
src/inference/yolov11_inference.py          28     11    61%   30-57
src/services/inference_service.py           63     19    70%   19-31, 55, 62-64, 77, 92-94
src/services/model_service.py               39      2    95%   17-18
src/services/test_inference_service.py      54      0   100%
src/services/test_model_service.py          35      0   100%
src/utils/file_utils.py                     13      0   100%
----------------------------------------------------------------------
TOTAL                                      469     60    87%
====================== 17 passed, 16 warnings in 8.67s =======================
```

Figure 96: Test Coverage Report for the Inference API

II 5.8.2 Examples of Tests

Below (Code-Fragment 31) is an example test for the `/models` endpoint, which lists available models. This test ensures that the endpoint responds correctly when the model service returns valid data and handles exceptions gracefully when the service fails.

Test for `/models` endpoint that mocks the model service dependency.

```python
from fastapi.testclient import TestClient
from unittest.mock import MagicMock
from src.api.endpoints import router, get_model_service
from fastapi import FastAPI

app = FastAPI()
app.include_router(router)
client = TestClient(app)

# Test the GET /models endpoint
def test_list_models():
    # Mock the model_service function
    mock_model_service = MagicMock(
        return_value=[{"name": "test_model", "latest_versions": []}]
    )
    # Override the dependency
    app.dependency_overrides[get_model_service] = lambda: mock_model_service

    response = client.get("/models")

    # Check the response
    assert response.status_code == 200
    assert response.json() == [{"name": "test_model", "latest_versions": []}]

    # Cleanup the dependency override
    app.dependency_overrides = {}
```

Code-Fragment 31: Test for `/models` endpoint that mocks the model service dependency.

The test:
1. Mocks the `model_service` dependency to simulate a valid response.
2. Calls the `/models` endpoint using a FastAPI test client.
3. Asserts that the status code and response body match expectations.

Another critical test is for the `/predict` endpoint, which processes an image and returns predictions (Code-Fragment 32):

Test for `/predict` endpoint simulating a prediction result.

```python
def test_predict():
    # Mock the interface_service function
    mock_interface_service = MagicMock(
        return_value=[{
            "x_min": 0, "y_min": 0, "x_max": 1, "y_max": 1,
            "score": 0.9, "label": "test",
        }]
    )
    # Override the dependency
    app.dependency_overrides[get_interface_service] = lambda: mock_interface_service

    # Create an in-memory "image" file
    image_file = BytesIO(b"fake image content")
    image_file.name = "test_image.jpg"

    query_params = {
        "name": "FasterRCNN_Model",
        "version": "2",
        "confidence_threshold": "0.5",
        "slice_height": "1024", "slice_width": "1024",
        "overlap_height_ratio": "0.2", "overlap_width_ratio": "0.2",
    }

    response = client.post(
        "/predict",
        files={"image": ("test_image.jpg", image_file, "image/jpeg")},
        params=query_params,
    )

    assert response.status_code == 200
    assert response.json() == {
        "predictions": [{
            "x_min": 0, "y_min": 0, "x_max": 1, "y_max": 1,
            "score": 0.9, "label": "test",
        }]
    }

    # Cleanup the dependency override
    app.dependency_overrides = {}
```

Code-Fragment 32: Test for `/predict` endpoint simulating a prediction result.

This test:
1. Mocks the `interface_service` dependency to simulate a prediction result.
2. Sends an in-memory image file along with prediction options as query parameters.
3. Asserts that the response contains the expected predictions.

II 5.8.3 Summary of Tests

The testing for the inference API focuses on the following key areas:
- **Endpoint Validation**: All endpoints are tested for both standard and edge-case inputs.
- **Model Management**: Verifies that models are correctly loaded, cached, and executed.
- **Error Handling**: Ensures that the API responds gracefully to exceptions, returning meaningful error messages.
- **Integration**: Validates communication between the inference API, backend, and frontend.
- **Performance**: Measures response times and scalability for inference tasks.

The test suite achieves high coverage and ensures that the inference API meets the functional and non-functional requirements of the project.

II 5.9 Backend API Testing

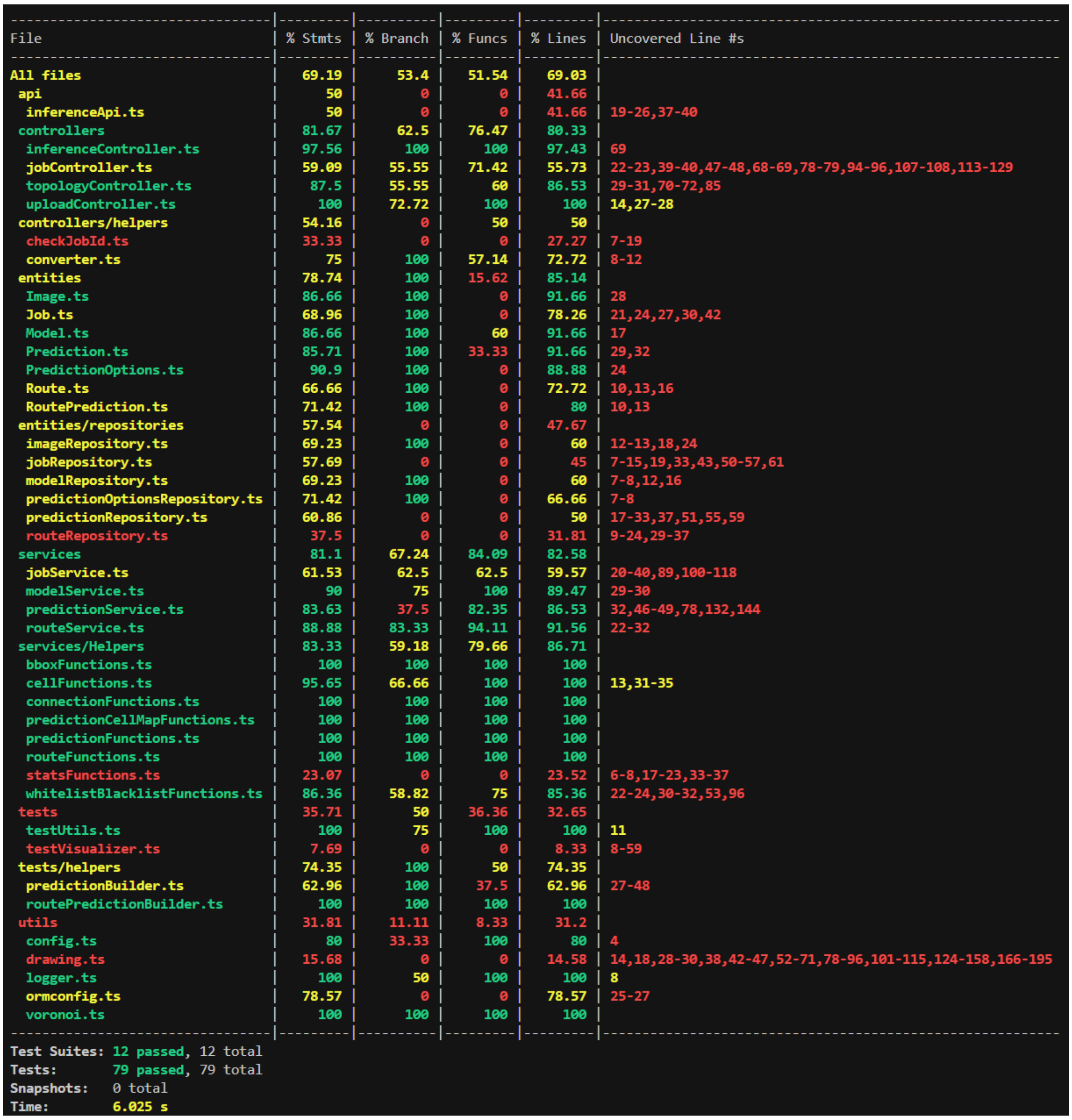
The **backend API** is a critical component of the BoltFinder system, responsible for processing requests, communicating with the inference API, managing the database, and serving data to the frontend. Extensive testing is conducted to validate its functionality, integration with other components, and error handling.

II 5.9.1 Overview of Testing

Tests are written using the Jest testing library. The backend API testing focused on the following areas:

1. **Service Testing**:
   - The core logic in the backend is encapsulated within services (e.g., `routeService`, `routeServiceEdit`, and `routeServicePanorama`), which handle operations such as route creation, editing, and panoramic transformations. These services are tested in isolation to ensure correctness.
2. **Integration Testing**:
   - Validates the interaction between backend services and external dependencies such as the inference API, database, and filesystem.
3. **Endpoint Testing**:
   - Ensures that all API endpoints function correctly, returning expected responses for standard and edge-case scenarios.

The screenshot in Figure 97 below illustrates the **test coverage** for the backend API, demonstrating comprehensive testing of all critical paths:



```
---------------------------------|---------|----------|---------|---------|------------------------------------------------------------
File                             | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
---------------------------------|---------|----------|---------|---------|------------------------------------------------------------
All files                        |   69.19 |     53.4 |   51.54 |   69.03 |
 api                             |      50 |        0 |       0 |   41.66 |
  inferenceApi.ts                |      50 |        0 |       0 |   41.66 | 19-26,37-40
 controllers                     |   81.67 |     62.5 |   76.47 |   80.33 |
  inferenceController.ts         |   97.56 |      100 |     100 |   97.43 | 69
  jobController.ts               |   59.09 |    55.55 |   71.42 |   55.73 | 22-23,39-40,47-48,68-69,78-79,94-96,107-108,113-129
  topologyController.ts          |    87.5 |    55.55 |      60 |   86.53 | 29-31,70-72,85
  uploadController.ts            |     100 |    72.72 |     100 |     100 | 14,27-28
 controllers/helpers             |   54.16 |        0 |      50 |      50 |
  checkJobId.ts                  |   33.33 |        0 |       0 |   27.27 | 7-19
  converter.ts                   |      75 |      100 |   57.14 |   72.72 | 8-12
 entities                        |   78.74 |      100 |   15.62 |   85.14 |
  Image.ts                       |   86.66 |      100 |       0 |   91.66 | 28
  Job.ts                         |   68.96 |      100 |       0 |   78.26 | 21,24,27,30,42
  Model.ts                       |   86.66 |      100 |      60 |   91.66 | 17
  Prediction.ts                  |   85.71 |      100 |   33.33 |   91.66 | 29,32
  PredictionOptions.ts           |    90.9 |      100 |       0 |   88.88 | 24
  Route.ts                       |   66.66 |      100 |       0 |   72.72 | 10,13,16
  RoutePrediction.ts             |   71.42 |      100 |       0 |      80 | 10,13
 entities/repositories           |   57.54 |        0 |       0 |   47.67 |
  imageRepository.ts             |   69.23 |      100 |       0 |      60 | 12-13,18,24
  jobRepository.ts               |   57.69 |        0 |       0 |      45 | 7-15,19,33,43,50-57,61
  modelRepository.ts             |   69.23 |      100 |       0 |      60 | 7-8,12,16
  predictionOptionsRepository.ts |   71.42 |      100 |       0 |   66.66 | 7-8
  predictionRepository.ts        |   60.86 |        0 |       0 |      50 | 17-33,37,51,55,59
  routeRepository.ts             |    37.5 |        0 |       0 |   31.81 | 9-24,29-37
 services                        |    81.1 |    67.24 |   84.09 |   82.58 |
  jobService.ts                  |   61.53 |     62.5 |    62.5 |   59.57 | 20-40,89,100-118
  modelService.ts                |      90 |       75 |     100 |   89.47 | 29-30
  predictionService.ts           |   83.63 |     37.5 |   82.35 |   86.53 | 32,46-49,78,132,144
  routeService.ts                |   88.88 |    83.33 |   94.11 |   91.56 | 22-32
 services/Helpers                |   83.33 |    59.18 |   79.66 |   86.71 |
  bboxFunctions.ts               |     100 |      100 |     100 |     100 |
  cellFunctions.ts               |   95.65 |    66.66 |     100 |     100 | 13,31-35
  connectionFunctions.ts         |     100 |      100 |     100 |     100 |
  predictionCellMapFunctions.ts  |     100 |      100 |     100 |     100 |
  predictionFunctions.ts         |     100 |      100 |     100 |     100 |
  routeFunctions.ts              |     100 |      100 |     100 |     100 |
  statsFunctions.ts              |   23.07 |        0 |       0 |   23.52 | 6-8,17-23,33-37
  whitelistBlacklistFunctions.ts |   86.36 |    58.82 |      75 |   85.36 | 22-24,30-32,53,96
 tests                           |   35.71 |       50 |   36.36 |   32.65 |
  testUtils.ts                   |     100 |       75 |     100 |     100 | 11
  testVisualizer.ts              |    7.69 |        0 |       0 |    8.33 | 8-59
 tests/helpers                   |   74.35 |      100 |      50 |   74.35 |
  predictionBuilder.ts           |   62.96 |      100 |    37.5 |   62.96 | 27-48
  routePredictionBuilder.ts      |     100 |      100 |     100 |     100 |
 utils                           |   31.81 |    11.11 |    8.33 |    31.2 |
  config.ts                      |      80 |    33.33 |     100 |      80 | 4
  drawing.ts                     |   15.68 |        0 |       0 |   14.58 | 14,18,28-30,38,42-47,52-71,78-96,101-115,124-158,166-195
  logger.ts                      |     100 |       50 |     100 |     100 | 8
  ormconfig.ts                   |   78.57 |        0 |       0 |   78.57 | 25-27
  voronoi.ts                     |     100 |      100 |     100 |     100 |
---------------------------------|---------|----------|---------|---------|------------------------------------------------------------
Test Suites: 12 passed, 12 total
Tests:       79 passed, 79 total
Snapshots:   0 total
Time:        6.025 s
```

Figure 97: Test Coverage Report for the Backend API

II 5.9.2 routeService Tests

The **routeService** is responsible for managing climbing routes, including their creation and deletion. The tests for this service validate its core functionality and error handling. The following Figure 97 shows the visualization of the test route scenarios:

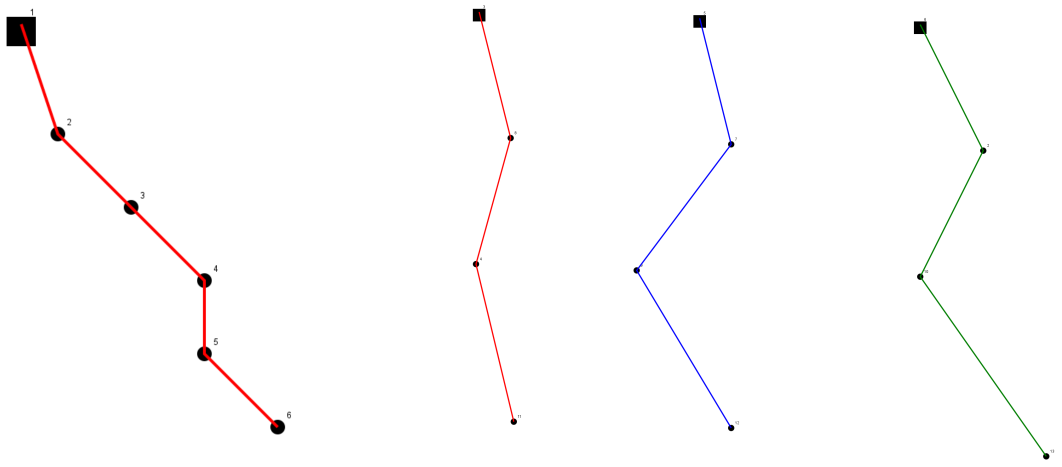


Figure 98: Visualization of two route scenarios

Below in Code-Fragment 33 is an example test for route creation:

Test for creating a new route using routeService.

```
import { routeService } from '../services/routeService';
import { mockDatabase } from '../mocks/mockDatabase';

describe('Route Creation', () => {
    it('should create a new route and return its details', async () => {
        const newRoute = {
            name: 'Test Route',
            difficulty: '5.10a',
            bolts: [{ x: 10, y: 20 }, { x: 15, y: 25 }]
        };

        const result = await routeService.createRoute(newRoute, mockDatabase);

        expect(result).toHaveProperty('id');
        expect(result.name).toBe(newRoute.name);
        expect(result.difficulty).toBe(newRoute.difficulty);
        expect(result.bolts).toEqual(newRoute.bolts);
    });
});
```

Code-Fragment 33: Test for creating a new route using routeService.

II 5.9.3 routeServiceEdit Tests

The **routeServiceEdit** handles operations related to modifying existing routes. This includes updating route details, repositioning bolts, or merging routes. The tests for this service ensure that route edits are processed correctly and consistently. The visualization in Figure 99 and Figure 100 show four of the scenarios:

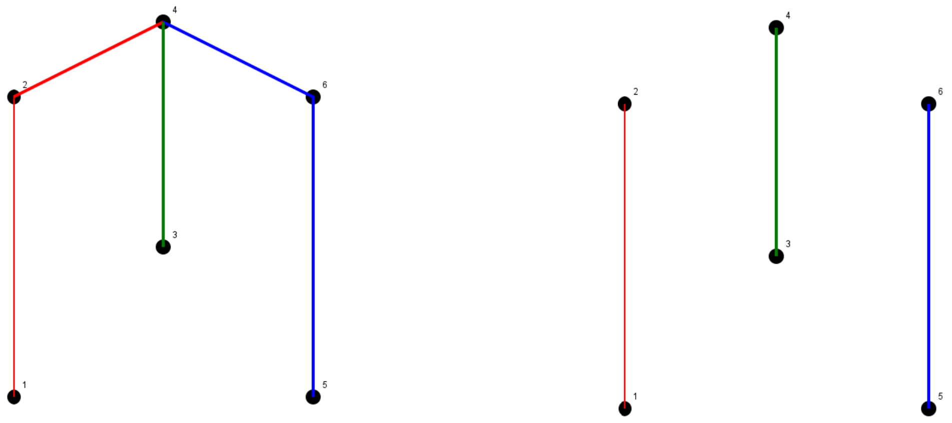


Figure 99: Visualization of two route editing scenarios

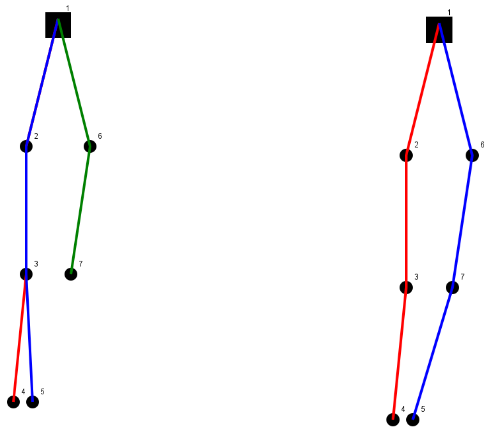


Figure 100: Visualization of two route editing scenarios

Below in Code-Fragment 34 is an example test for editing route details:

Test for updating route details using routeServiceEdit.

```
import { routeServiceEdit } from '../services/routeServiceEdit';
import { mockDb } from '../mocks/mockDatabase';

describe('Route Editing', () => {
  it('should update route details and return the updated route', async () => {
    const existingRouteId = '12345';
    const updatedDetails = {
      name: 'Updated Route Name',
      difficulty: '5.11b'
    };

    const result = await routeServiceEdit.updateRoute(existingRouteId, updatedDetails, mockDb);

    expect(result.id).toBe(existingRouteId);
    expect(result.name).toBe(updatedDetails.name);
    expect(result.difficulty).toBe(updatedDetails.difficulty);
  });
});
```

Code-Fragment 34: Test for Updating Route Details using routeServiceEdit.

II 5.9.4 routeServicePanorama Tests

The **routeServicePanorama** processes panoramic climbing wall images, creating seamless routes that span across multiple sections of the wall. Testing for this service focused on verifying transformations, consistency of routes, and error handling for invalid panoramic inputs. The visualization in Figure 101 shows visualization of the panoramic route test.

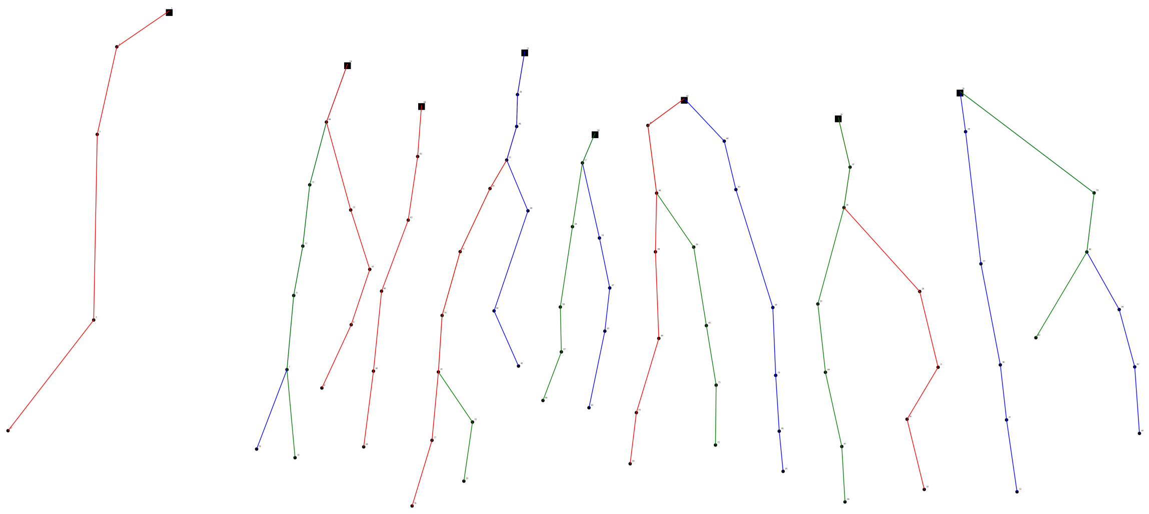


Figure 101: Visualization of a panoramic climbing wall, showing routes across multiple wall sections.

Below in Code-Fragment 35 is an example test for processing panoramic routes:

Test for generating panoramic routes using routeServicePanorama.

```
import { routeServicePanorama } from '../services/routeServicePanorama';
import { mockPanoramaInput } from '../mocks/mockPanoramaInput';

describe('Panoramic Route Generation', () => {
    it('should generate routes across a panoramic image and return details', async () => {
        const result = await routeServicePanorama.generateRoutes(mockPanoramaInput);

        expect(result).toHaveProperty('routes');
        expect(result.routes.length).toBeGreaterThan(0);
        expect(result.routes[0]).toHaveProperty('name');
    });

    it('should handle invalid panoramic input and throw an error', async () => {
        const invalidInput = null;

        await expect(routeServicePanorama.generateRoutes(invalidInput))
            .rejects
            .toThrow('Invalid panoramic input');
    });
});
```

Code-Fragment 35: Test for Generating Routes using routeServicePanorama.

II 5.9.5 Summary of Tests

The backend API tests cover the following areas:

- **Service Logic**: Each service (`routeService`, `routeServiceEdit`, `routeServicePanorama`) is tested to validate core functionality, error handling, and edge cases.
- **Integration**: Verifies seamless interaction between backend services, the inference API, and database.
- **Panoramic Processing**: Ensures that panoramic routes are generated accurately across large climbing wall images.

The comprehensive test suite ensures that the backend API meets the functional and non-functional requirements of the project.

II 5.10 Frontend Testing

The frontend testing is conducted through manual user tests to validate the functionality and usability of the system. Each test is documented below with its steps and expected outcomes.

II 5.10.1 User Test Cases

User test cases are designed to cover the primary functionalities of the frontend, including image upload, prediction, annotation editing, and route visualization. The following test cases are executed to validate the system's behavior.

II 5.10.1.1 Test Case 1: Upload

Step	Description
1	Navigate to the upload page.
2	Select an image to be uploaded.
3	Click the upload button.
Expected Outcome	A success message is displayed indicating the image is uploaded successfully, and a progress bar is visible during the upload process.
Result	Test passed: The success message is displayed as expected, and the progress bar is visible during the upload.

Table 22: Test Case 1: Upload

II 5.10.1.2 Test Case 2: Prediction

Step	Description
1	Prerequisite: An image has been uploaded.
2	Navigate to the prediction page.
3	Select a model, version, confidence, tile size, and overlap.
4	Submit the prediction job.
Expected Outcome	A success message is displayed indicating the prediction is successfully initiated. On the status page, the job status is initially "Pending" and switches to "Finished" once the prediction is complete.
Result	Test passed: The success message is displayed as expected, and the job status updates from "Pending" to "Finished" on the status page after completion.

Table 23: Test Case 2: Prediction

II 5.10.1.3 Test Case 3: Add New Bolt

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click "Edit" to be redirected to the edit page.
4	Click on the drawing tool and create a new annotation using the mouse.
5	Select the new annotation as a bolt and click save.
Expected Outcome	A success message is displayed indicating the bolt is successfully added.
Result	Test passed: The success message is displayed as expected.

Table 24: Test Case 3: Add New Bolt

II 5.10.1.4 Test Case 4: Adjust Bolt

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click “Edit” to be redirected to the edit page.
4	Select an existing bolt annotation and move it to a new position.
5	Click save.
Expected Outcome	A success message is displayed indicating the bolt position is successfully updated.
Result	Test passed: The success message is displayed as expected.

Table 25: Test Case 4: Adjust Bolt

II 5.10.1.5 Test Case 5: Remove Route

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click “Edit” to be redirected to the edit page.
4	In the route table, select a route and click delete.
5	Click save.
Expected Outcome	A success message is displayed indicating the route is successfully removed.
Result	Test passed: The success message is displayed as expected.

Table 26: Test Case 5: Remove Route

II 5.10.1.6 Test Case 6: Edit Route

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click “Edit” to be redirected to the edit page.
4	In the route table, select a route to edit.
5	In the viewer, select an annotation not part of the route and add it to the route via the route table.
6	Click save.
Expected Outcome	A success message is displayed indicating the route is successfully updated.
Result	Test passed: The success message is displayed as expected.

Table 27: Test Case 6: Edit Route

II 5.10.1.7 Test Case 7: Map Routes on Image

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click “Download Topo.”
Expected Outcome	A success message is displayed, and an image with the routes drawn is downloaded successfully.
Result	Test passed: The success message is displayed as expected.

Table 28: Test Case 7: Map Routes on Image

II 5.10.1.8 Test Case 8: Delete Job

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click the “Delete” button.
Expected Outcome	A confirmation modal is displayed before deletion is executed.
Result	Test failed: No confirmation modal is displayed for deleting a job.

Table 29: Test Case 8: Delete Job

II 5.10.1.9 Test Case 9: Recalculate Routes from Scratch

Step	Description
1	Prerequisite: An image is uploaded, and the prediction job is finished.
2	Navigate to the status page and select a job.
3	Click “Edit” to be redirected to the edit page.
4	Click the “Recalculate” button.
Expected Outcome	A confirmation modal is displayed before recalculation is executed.
Result	Test passed: The confirmation modal is displayed as expected.

Table 30: Test Case 9: Recalculate Routes from Scratch

II 5.10.2 Conclusion

The frontend testing validates the key functionalities through manual user tests. All features are successfully tested, except for the “Delete Job” test, where the absence of a confirmation modal indicates a failure. Further refinement is required to address this issue and ensure consistent handling of destructive user actions.

II 6 Results and Further Development

This chapter presents the results of the different technical implementations as well as the model comparison and performance.

II 6.1 Results

The results of the different implementations are shown in the following sections. Furthermore the achievement of functional and non-functional requirements is documented in Chapter II 6.1.3.

II 6.1.1 Comparison of Models

Multiple model versions with different tile sizes are trained and compared in this chapter. The Table 31 shows the different model versions and their performance. Other versions are described in Chapter II 5.2.3.

Model	Training Tile Size	Run	Model Version	Dataset Version
Faster R-CNN	800px	Run 11	v7	v11
YOLOv11	800px	Run 11	v21	v11
YOLOv11	512px	Run 17	v31	v17-512
YOLOv11	800px	Run 17	v28	v17-800
YOLOv11	1024px	Run 17	v32	v17-1024
YOLOv11	1024px	Run 18	v33	v18

Table 31: Model Versions

- The Faster R-CNN model is compared to the YOLOv11 model in Chapter II 6.1.1.1.
- The best tile size for the YOLOv11 model is compared in Chapter II 6.1.1.2.
- The performance of the combined model is shown in Chapter II 6.1.1.4.

II 6.1.1.1 Faster R-CNN vs YOLOv11 Model

By comparing a run with the same dataset of the Faster R-CNN and YOLOv11 model via the metrics described in Chapter I 3, the YOLOv11 model proves to be the better fitting model for detecting bolts and anchors in the high resolution panorama images. The comparison of the models is shown in the following figures.

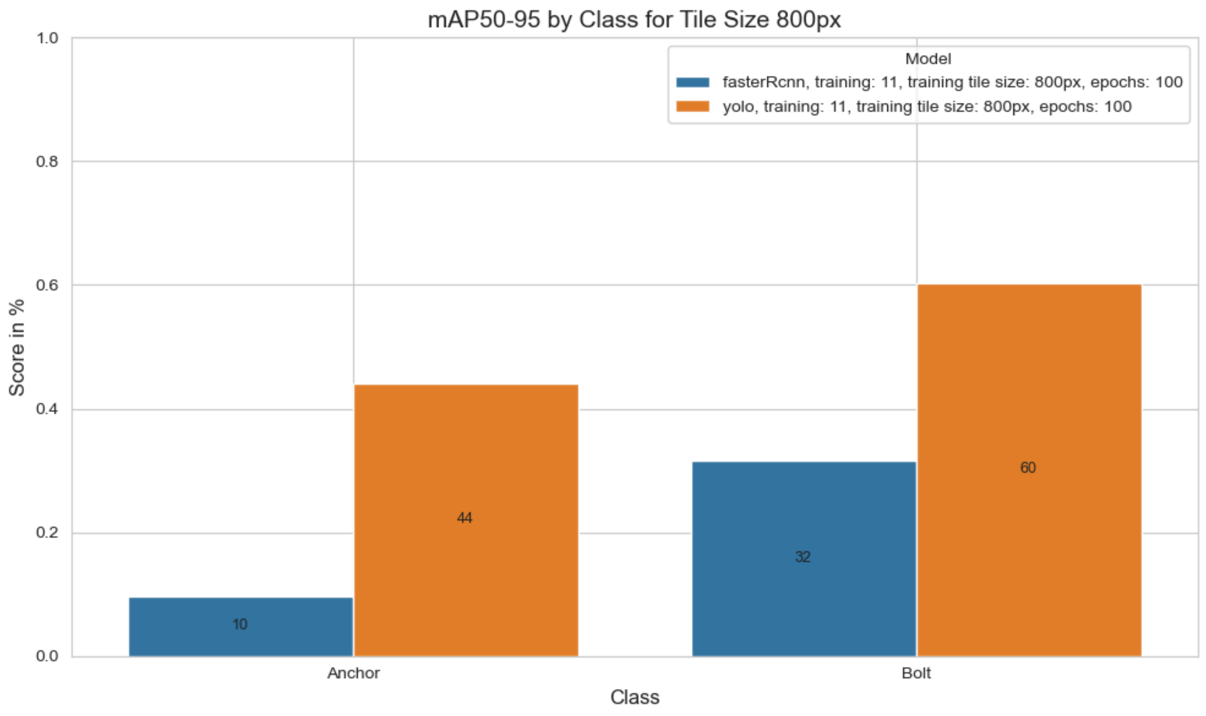


Figure 102: Faster R-CNN vs YOLOv11 mAP50-95 Comparison

The results depicted in Figure 102 indicate that the YOLOv11 model outperforms the Faster R-CNN model in terms of mAP50-95 across both classes. Similarly, as shown in Figure 103, YOLOv11 achieves higher F1 scores for both classes.

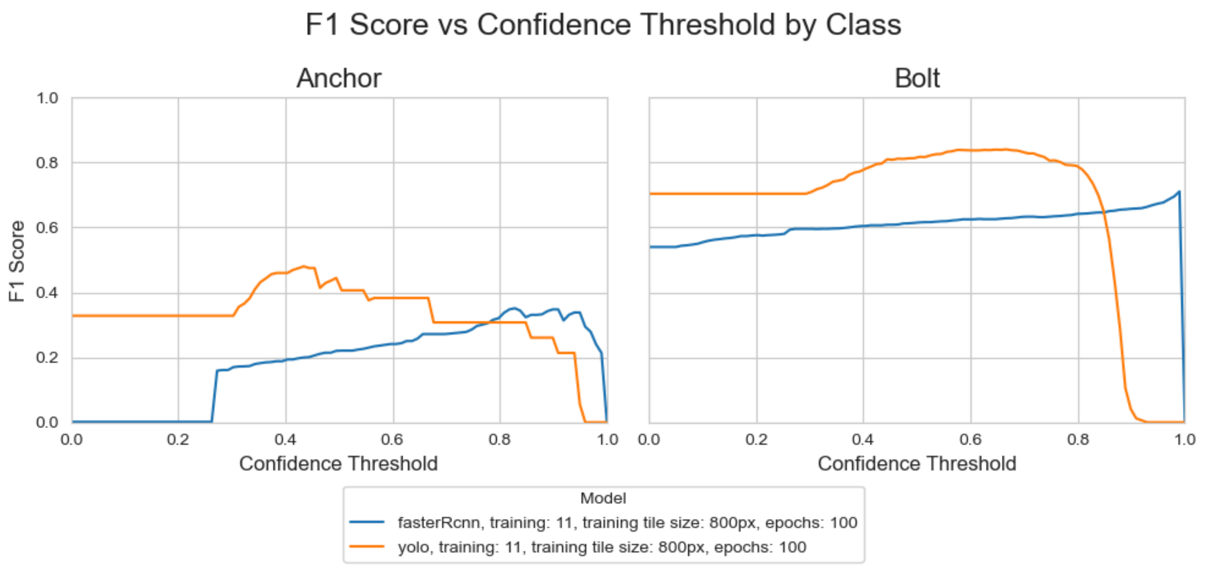


Figure 103: Faster R-CNN vs YOLOv11 F1 Comparison

Although the Faster R-CNN model surpasses YOLOv11 in F1 score at confidence levels above approximately 0.82, Figure 104 reveals that Faster R-CNN exhibits lower precision. This suggests that the Faster R-CNN model is more prone to predicting false positives. Consequently, the higher precision of YOLOv11 makes it the preferred choice, as it reduces the number of false-positive bolts and anchors that users would need to manually delete.

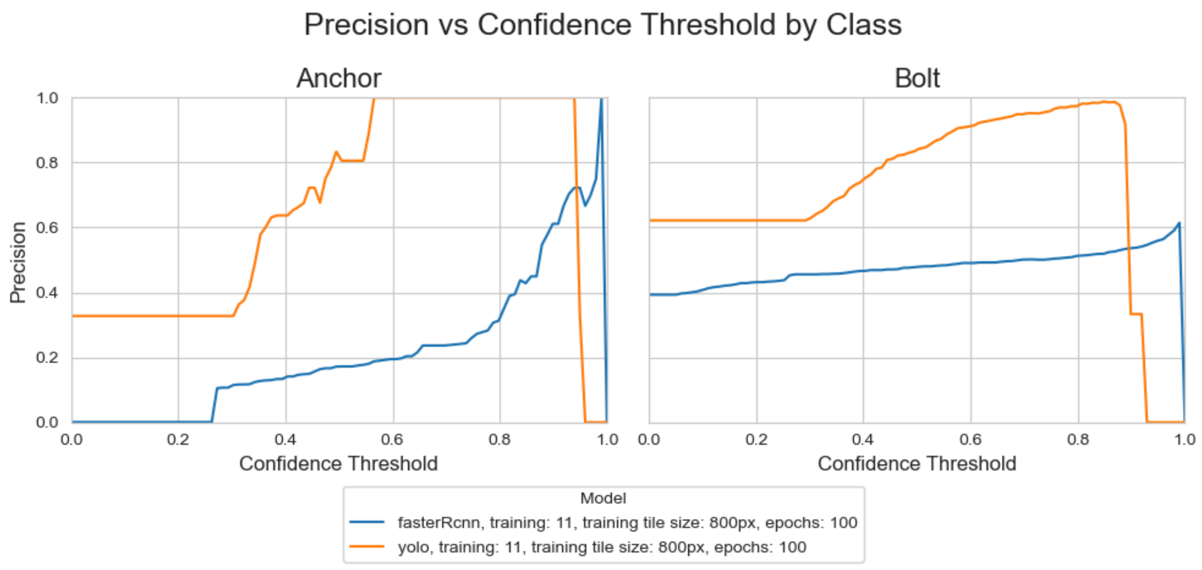


Figure 104: Faster R-CNN vs YOLOv11 Precision Comparison

More detailed results of the comparison between the Faster R-CNN and YOLOv11 models are provided in Chapter III 3 Figure 138

II 6.1.1.2 Best Tile Size

The following tile sizes are compared for training the YOLOv11 model: 512px, 800px, and 1024px. The optimal training and inference tile size for the YOLOv11 model is 1024px. The subsequent sections present a comparison of the various tile sizes.

Caution

The three training models are compared with the same raw training data, scaled and sliced to their specific training tile sizes. Please note that this is not exactly the same dataset anymore. Since the test data is also rather small, the values might not be generalizable to other datasets. More information about the training and test data, as well as dataset creation, can be found in Chapter II 5.1.

Smaller tile sizes have a higher chance of cutting the bolts and anchors bounding boxes. Therefore, the testing is conducted with a 60% inference overlap. With this in mind the testing is done with a 60% inference overlap.

Note

The tile size 2048px is not compared, because the training runs out of memory and is not possible on the projects training infrastructure. This error persists with batch sizes from 4 to 12. The error message is shown in the following Code-Fragment 36

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 48.00 MiB. GPU 0 has
a total capacity of 15.89 GiB of which 22.12 MiB is free. Process 2340680 has
15.86 GiB memory in use. Of the allocated memory 15.47 GiB is allocated by
PyTorch, and 46.49 MiB is reserved by PyTorch but unallocated. If reserved but
unallocated memory is large try setting max_split_size_mb to avoid fragmentation.
See documentation for Memory Management and PYTORCH_CUDA_ALLOC_CONF
```

Code-Fragment 36: OutOfMemoryError on DGX-1

II 6.1.1.2.1 Comparing mAP metrics

The mAP metrics are compared for the different training tile sizes and inference tile sizes. The mAP metrics are shown in the following Figure 105. Over all IoU thresholds the 1024px tile size for training and inference has the best mAP scores. The mAP50-95 shows this nicely with the highest score of 0.45 for the 1024px tile size. The inference tile size of 2048 also has high mAP scores, but it also is less precise and predicts more false positives as shown later in Chapter II 6.1.1.2.2.

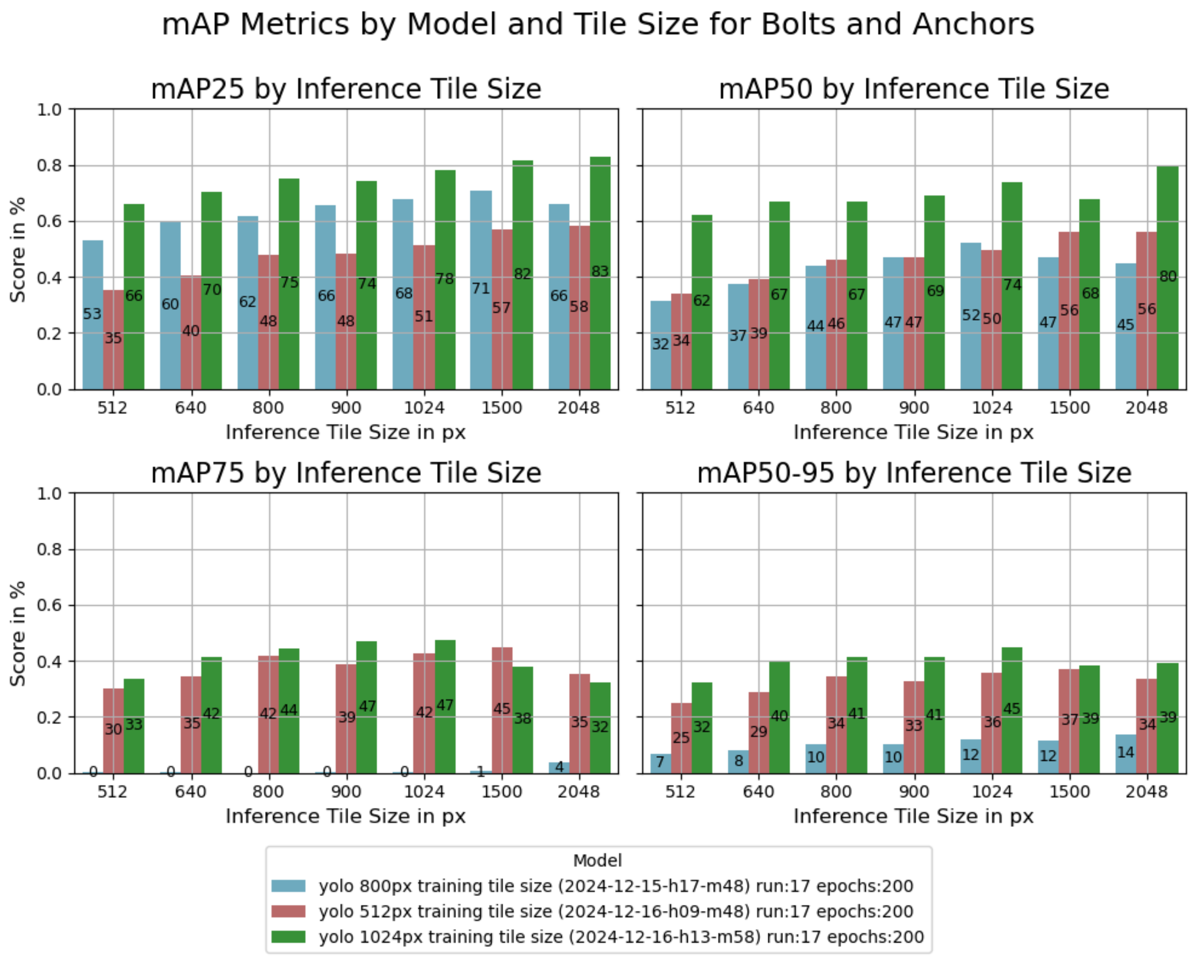


Figure 105: Tile Size Comparison for Bolts and Anchors

Another interesting fact is that the 800px training tile size has the worst mAP50-95 scores and even reaches 0 for the mAP75 while still being better than the 512px tile size for the mAP25. This can be explained when looking at the bounding boxes in Figure 106. The bolts are detected but ground truth annotation area is so small that even a small deviation in the bounding box can lead to a low IoU score. This means for this project the mAP25 is a better metric to compare the models, because the annotations don't have to be very precise.

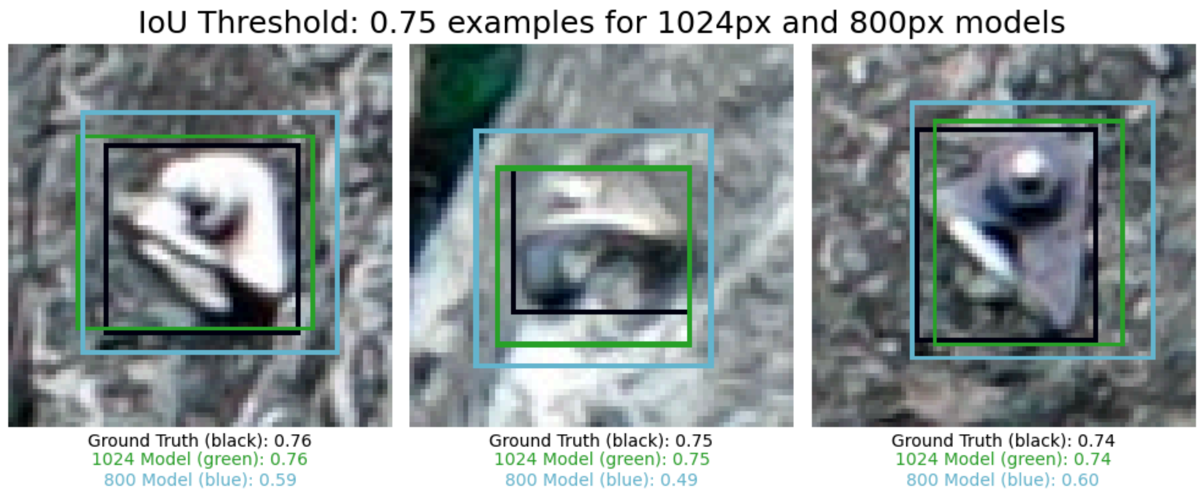


Figure 106: mAP75 precise bounding boxes problem

The mAP metrics by bolt and anchor are in Chapter III 3.2.

II 6.1.1.2.2 Comparing F1

The F1 metrics are compared per training tile size for different inference tile sizes. The metrics are shown in the following Figure 107.

II 6.1.1.2.2.1 F1 Score vs Confidence Threshold

When plotting the median F1 score per inference tile size over all three models (512px, 800px, 1024px) in Figure 107. The 1500px and 2048px perform the worst for bolts. The other inference tile sizes have a performance similar to each other. For anchors the values are more tangled and spread evenly.

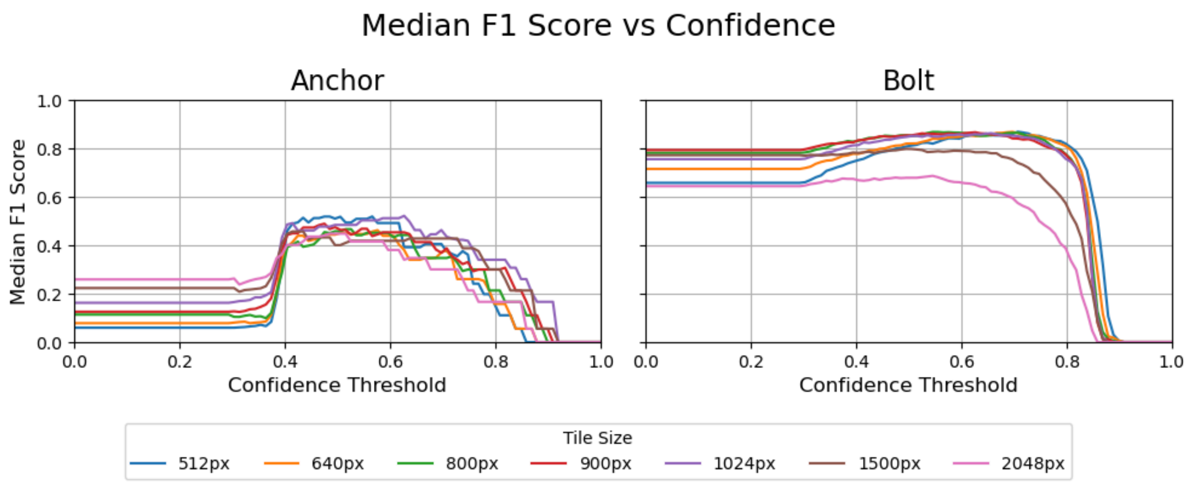


Figure 107: Median F1 Score vs Confidence Threshold

The inference tile sizes 512px up to 1024px are not easily comparable, because the F1 scores are very close to each other. By looking at the area under the curve (AUC) for the F1 score after a confidence threshold of 0.35 the 1024px inference tile size has the largest area for anchors with 0.23 and for bolts the area is 0.02 lower compared to smaller inference tile sizes. The AUC is shown in Figure 108.

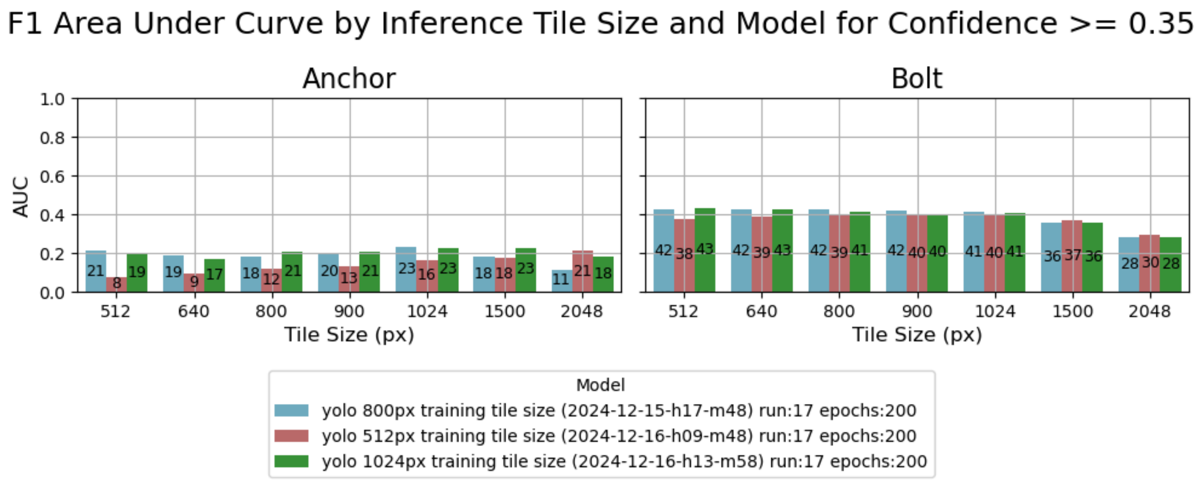


Figure 108: AUC F1 Score

Aggregating the F1 scores (mean, median & max) also shows that the 1500px and 2048px inference tile sizes perform the worst for bolts, but can be better for anchors (Figure 109, Figure 110, Figure 111).

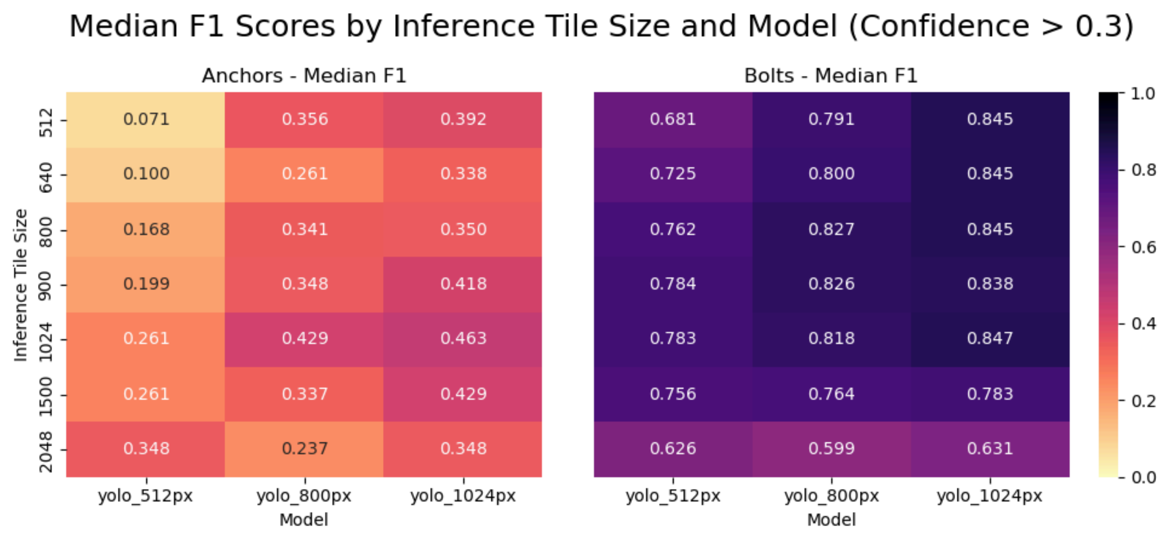


Figure 109: Aggregating Median F1 Scores at Confidence 0.3

The F1 score for inference tile sizes from 512px to 1024px are also in Figure 109, Figure 110, Figure 111 grouped together by a few percentages for bolts, but not for anchors. The anchors have much less training and test data, which can explain the higher variance in the F1 scores.

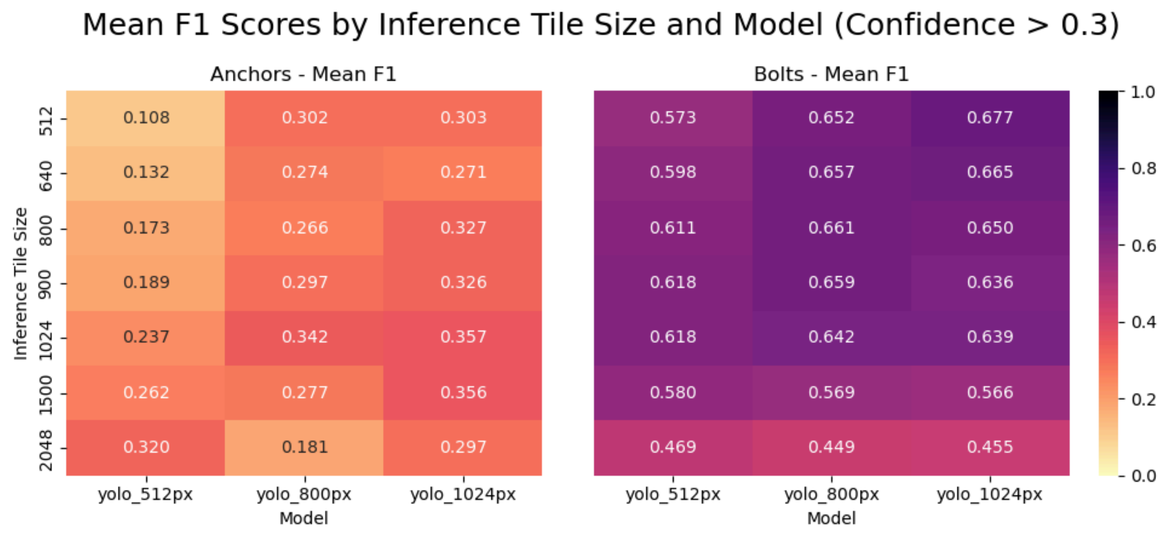


Figure 110: Aggregating Mean F1 Scores at Confidence 0.3

When looking at the maximum F1 scores depicted in Figure 111 the 512px model sometimes achieves higher scores. This is also shown in Figure 112 for the class bolt and tile size 1024 where the 512px model peaks at around 0.7 confidence while being worse at the 0.4 confidence level. The 1024px model is better at the 0.4 confidence level.

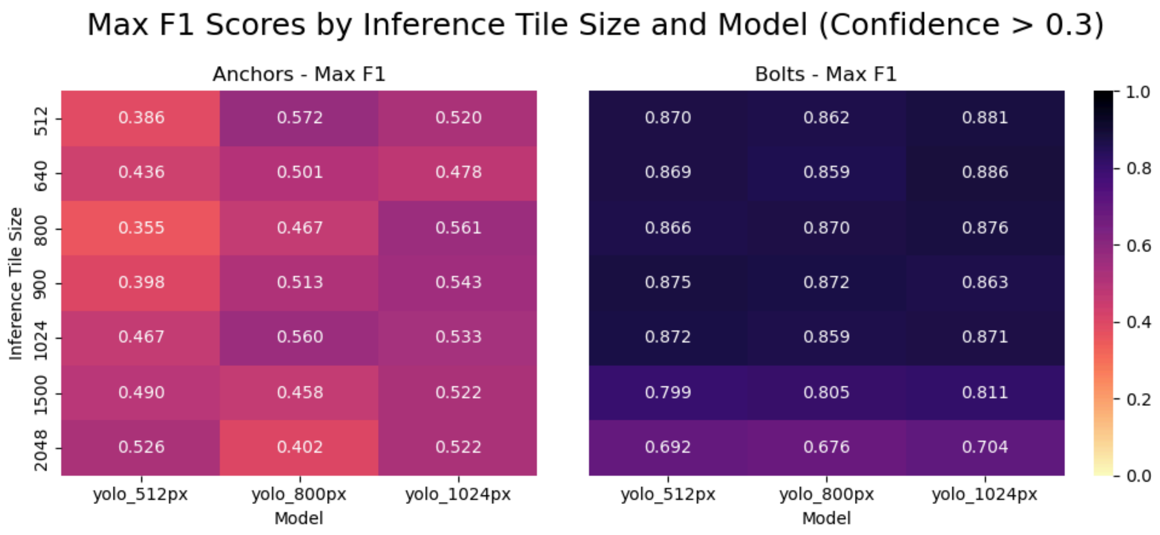


Figure 111: Aggregating Max F1 Scores at Confidence 0.3

Comparing the inference tile size 512px to 1024px shows these stats:
- the model trained with 512px performs worse for bolts and anchors at lower confidence thresholds.
- the model trained with 1024px performs better for bolts and anchors at lower confidence thresholds.
- the model trained with 800px reaches the highest F1 score for anchors.

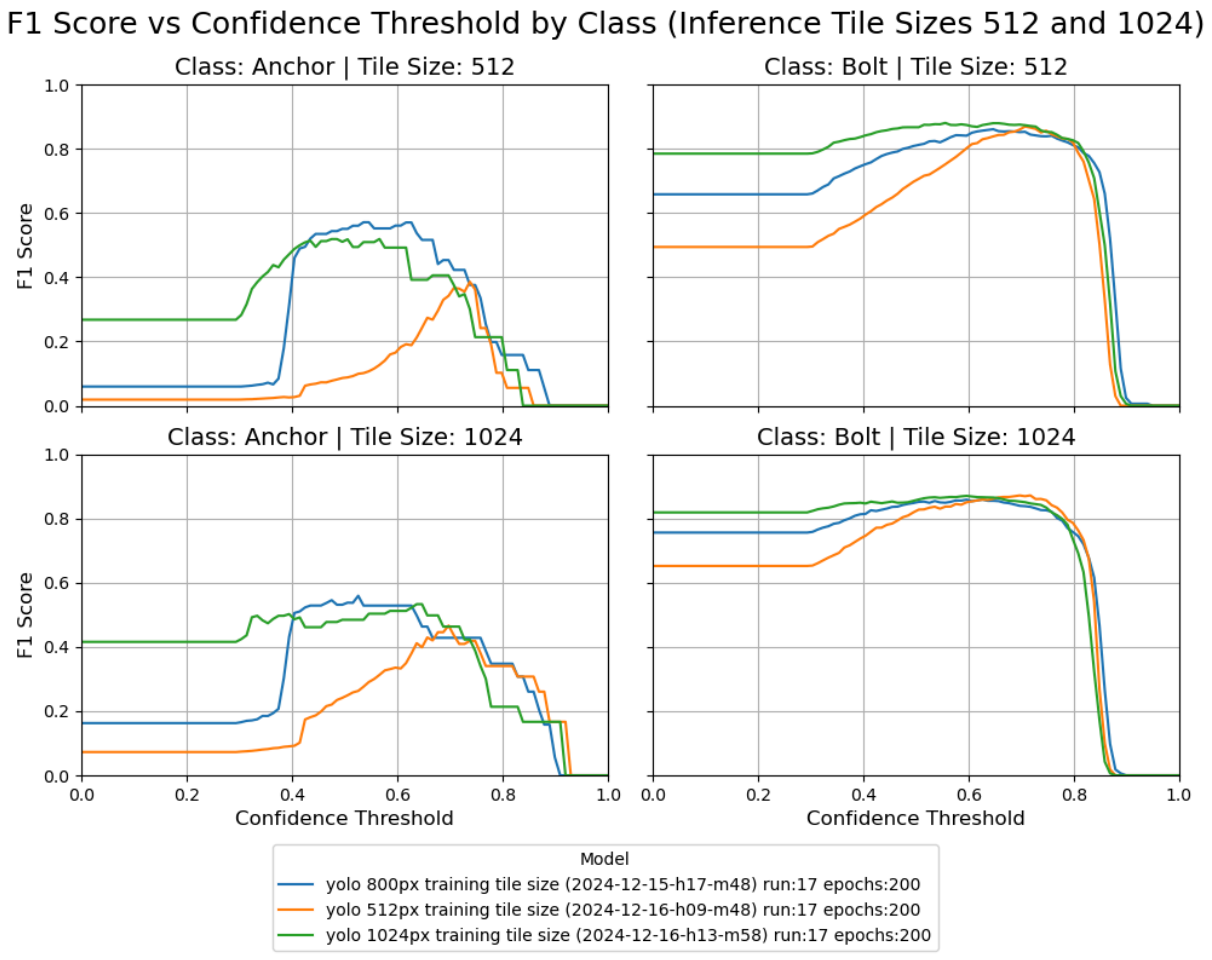


Figure 112: F1 Score vs Confidence Threshold by Class (Inference Tile Sizes 512 and 1024)

Additional plots are shown in Chapter III 3.2.

II 6.1.1.2.3 Best Tile Size Conclusion

By having the best results in the mAP metrics and performing early on good in the F1 metrics the 1024px training tile size is the best choice for training and inference. The 800px training tile size performs worse in the mAP metrics and the 512px training tile size performs worse in the F1 metrics. For the inference tile size the 1024px tile size is also the optimal choice. Other sizes can be better at certain confidence thresholds, but the 1024px tile size is the best overall choice.

II 6.1.1.3 F1, Precision and Recall for the Best Tile Size Model.

The F1, Precision and Recall metrics are shown for the model from run 17 with the training and inference tile size of 1024px. The metrics are shown in the following figures below.

II 6.1.1.3.1 F1 Metrics

In Figure 113 the aggregated F1 scores for the bolts and anchors together are displayed.

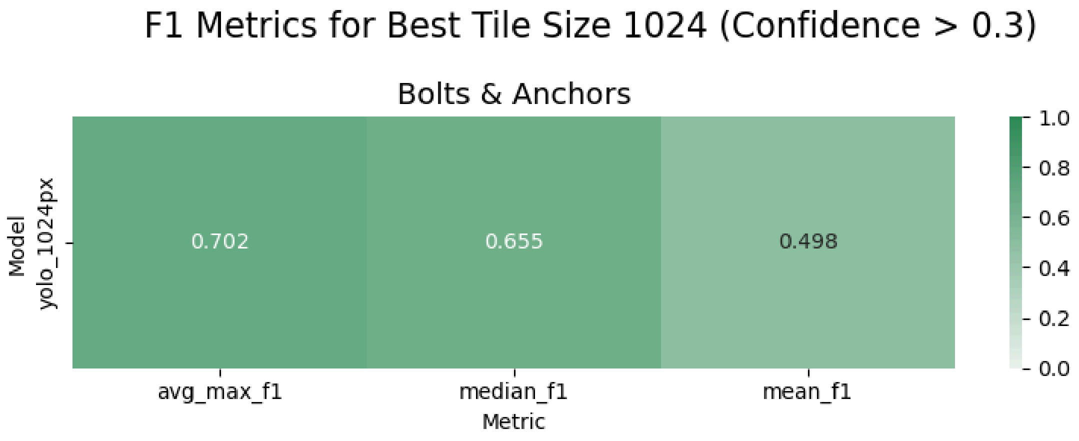


Figure 113: F1 Metrics for Best Tile Size 1024 at Confidence 0.3 (both classes)

In Figure 114 the F1 scores are shown by class. The model achieves a maximum F1 score of 87% for bolts and a maximum F1 score of 53% for anchors.

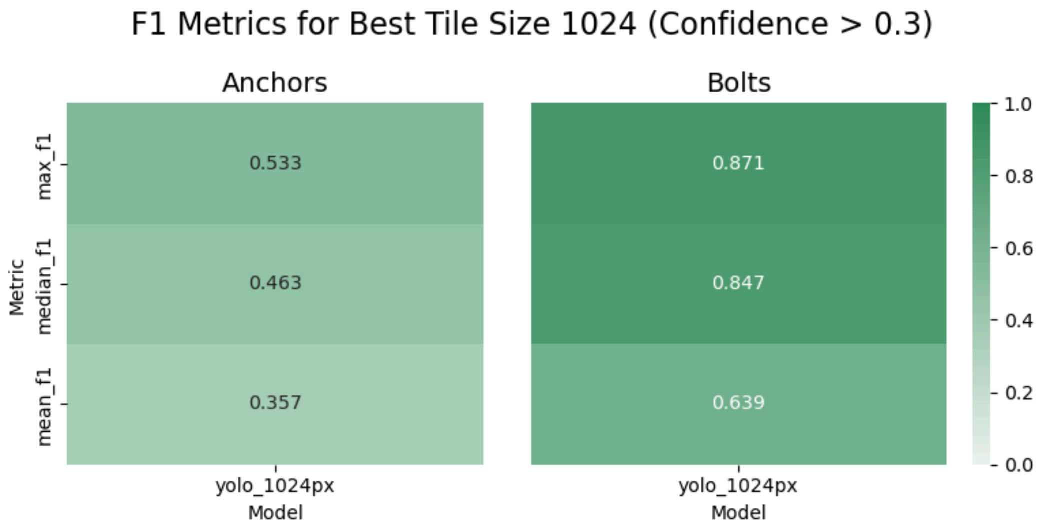


Figure 114: F1 Metrics for Best Tile Size 1024 at Confidence 0.3

In Figure 115 the F1 score is plotted against the confidence threshold. It shows that the model is better at detecting bolts than anchors. Also the model starts dropping in F1 score at a confidence of 60% for bolts and 64% for anchors.

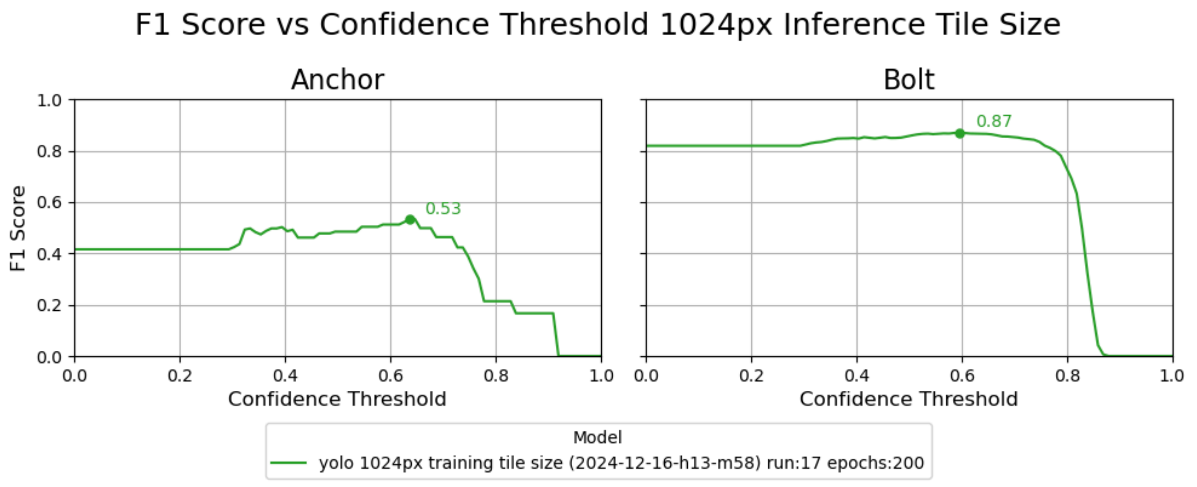


Figure 115: F1 Score vs Confidence Threshold (1024px Inference Tile Size)

II 6.1.1.3.1.1 Precision and Recall Metrics

The following two figures (Figure 116 and Figure 117) show the precision and recall metrics for the model with the best tile size of 1024px. The model achieves a high precision of 79% for bolts at a confidence of zero already, whereas the for anchors the precision starts at 39%. The maximum recall for bolts is 86% and for anchors 48%.

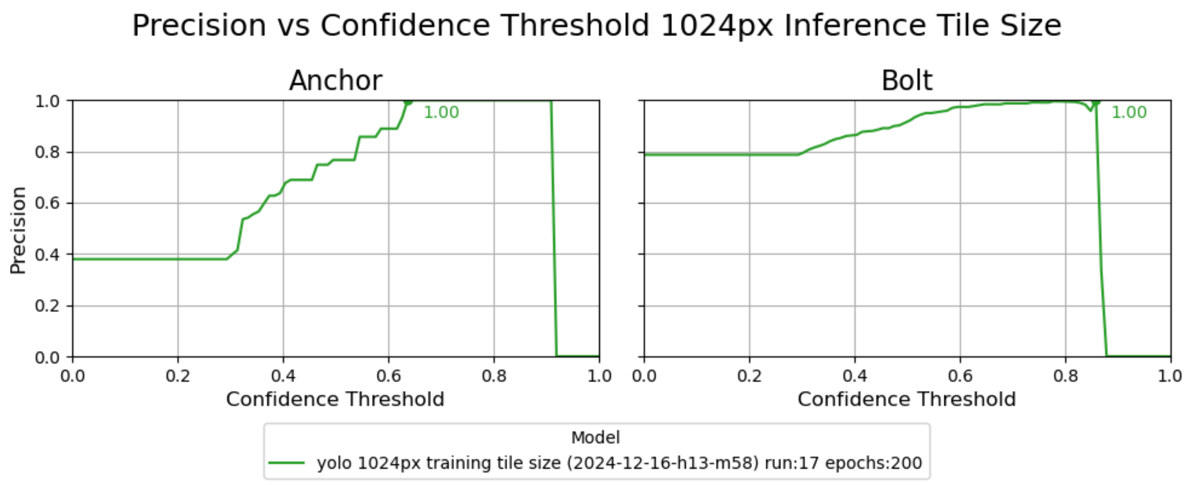


Figure 116: Precision vs Confidence Threshold (1024px Inference Tile Size)

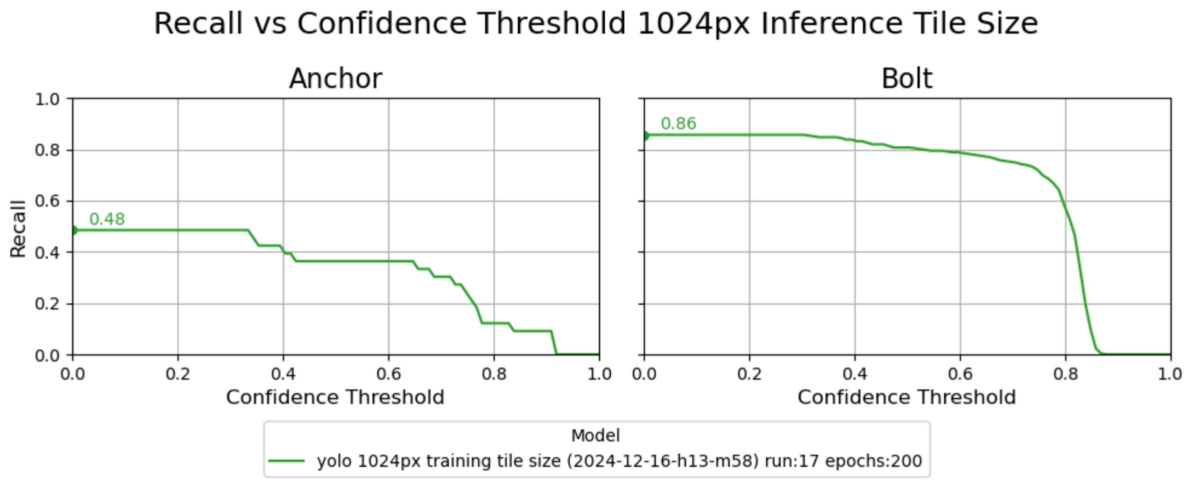


Figure 117: Recall vs Confidence Threshold (1024px Inference Tile Size)

II 6.1.1.4 Combined Model

With the training 18 model version 33 the training and test data is used with the best tile size of 1024px. This model is the best performing model and should be used for running predictions on panorama images.

> ⚠ **Warning**
>
> Because test data is used for training the model can familiarize itself with the test data and the inference results are not generalizable (see Chapter II 5.1.6.3). The model is therefor also tested on all the panorama images of the climbing sector "Chämiloch" in Schwyz.

II 6.1.1.4.1 mAP Scores

The mAP metrics for the combined model are shown in the following Figure 118. The mAP scores are better than all previously trained models.

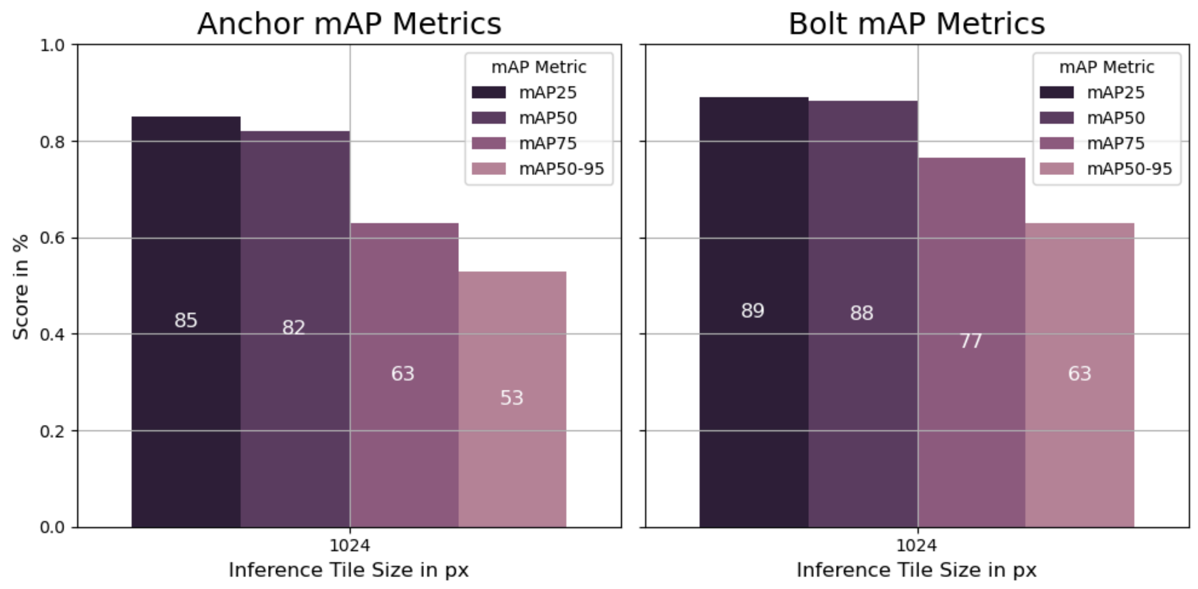


Figure 118: mAP Metrics for Combined Model

II 6.1.1.4.2 F1 Scores

The F1 scores for the combined model for bolts and anchors together are shown in the following Figure 119. The F1 scores are better than all previously trained models.

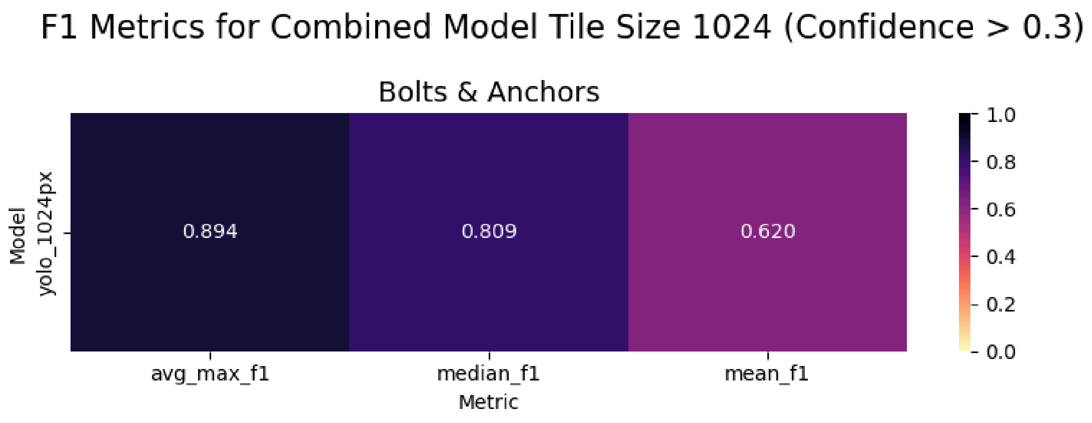


Figure 119: F1 Metrics for Combined Model without Split

The F1 scores for the combined model are shown in the following Figure 120. Especially for the class anchor the F1 score is significantly higher than for previously trained models. The F1 score for bolts is also starting higher than for previously trained models.

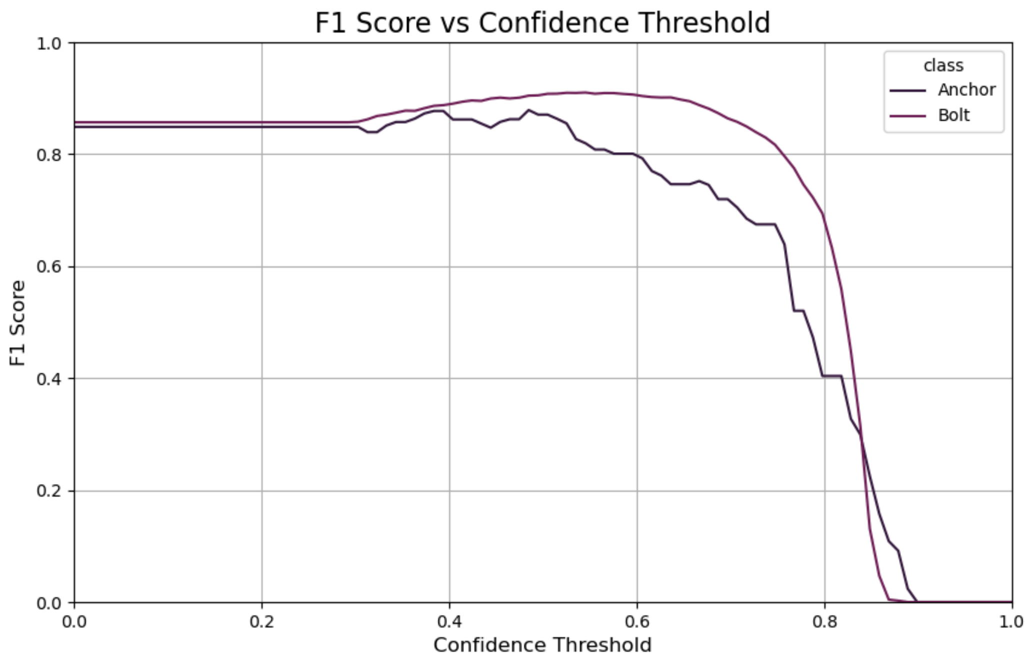


Figure 120: F1 vs Confidence Threshold for Combined Model

Looking at mean, median, and max F1 scores for the combined model in the following Figure 121 gives concrete values of the model's performance. The max F1 score is 91% for bolts and 87.9% for anchors.

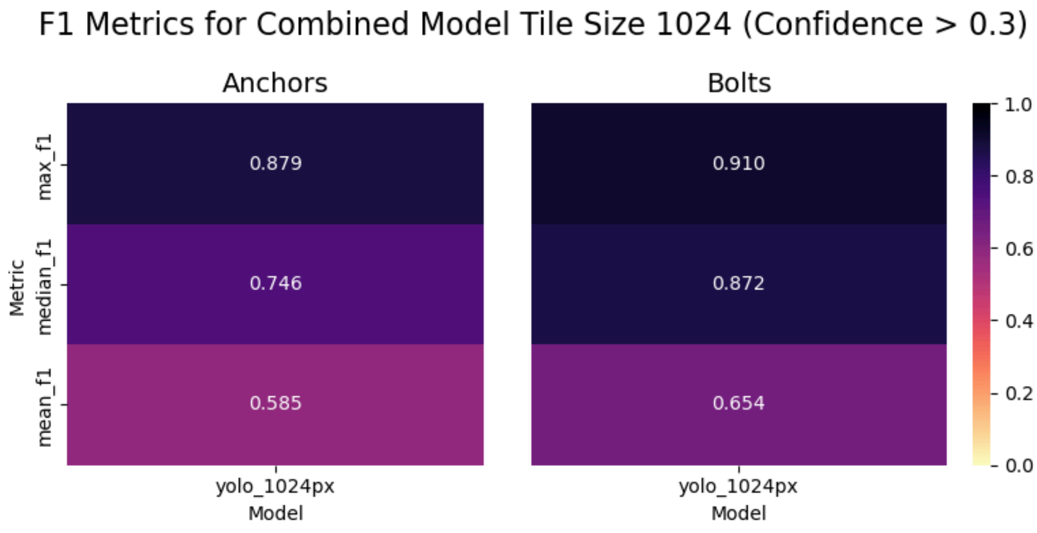


Figure 121: F1 Metrics for Combined Model

II 6.1.1.4.3 Precision and Recall

The precision and recall scores for the combined model are shown in the following Figure 122 and Figure 123. The precision and recall scores show that the model is predicting 90% of the bolts with a precision of 80% already at the confidence threshold of zero.

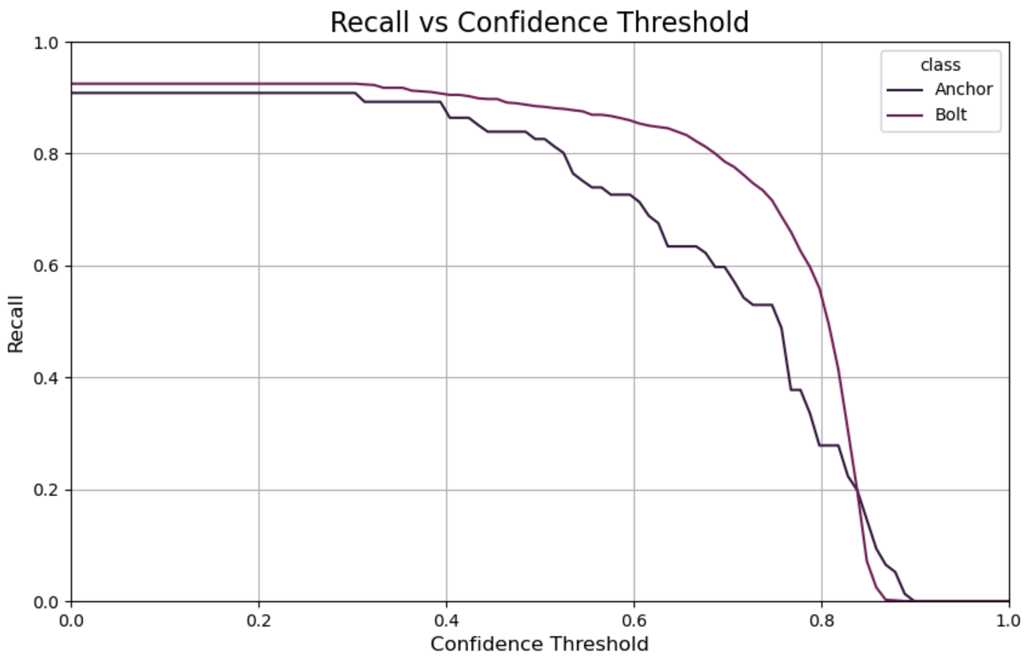


Figure 122: Recall for Combined Model

The precision reaches 100% at a confidence threshold of 0.9 for bolts and 0.7 for anchors.

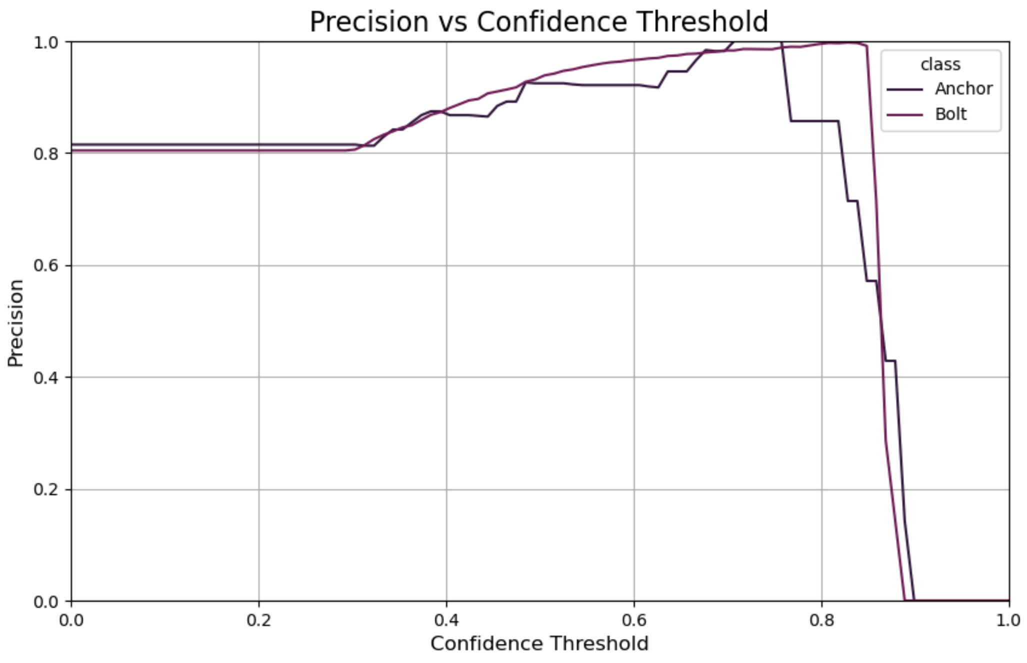


Figure 123: Precision for Combined Model

II 6.1.1.4.4 True Positives

The true positives in Figure 124 show that the model is able to detect most of the bolts and anchors in the panorama images. The confidence levels of the true positives are also shown in the image. The green bounding boxes are the ground truths and the red bounding boxes are the predictions of the model.

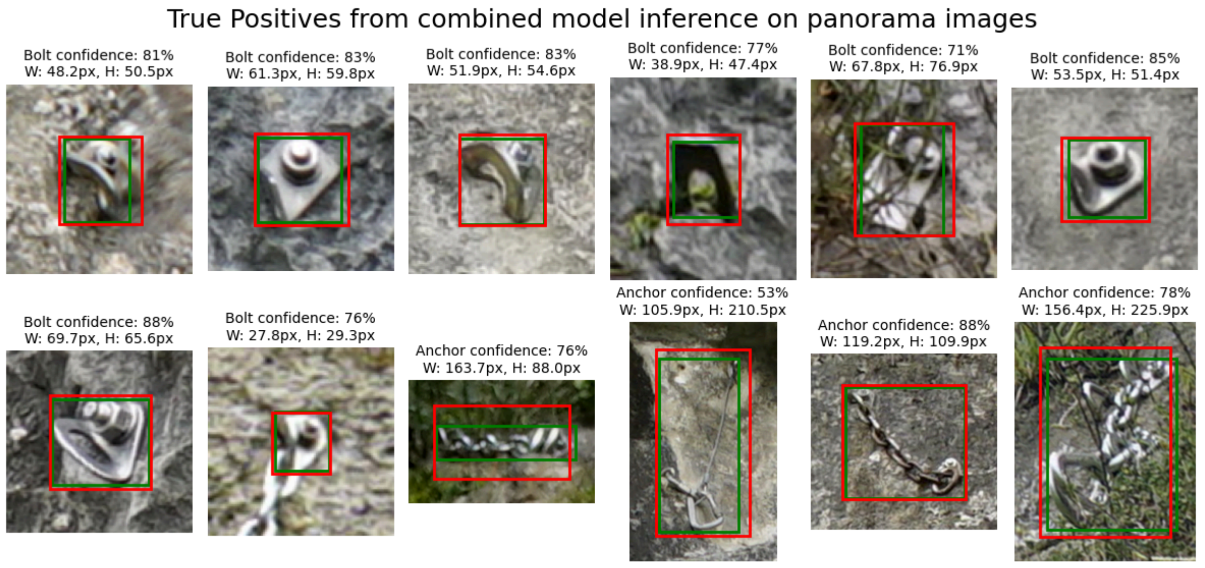

True Positives from combined model inference on panorama images
Bolt confidence: 81% W: 48.2px, H: 50.5px
Bolt confidence: 83% W: 61.3px, H: 59.8px
Bolt confidence: 83% W: 51.9px, H: 54.6px
Bolt confidence: 77% W: 38.9px, H: 47.4px
Bolt confidence: 71% W: 67.8px, H: 76.9px
Bolt confidence: 85% W: 53.5px, H: 51.4px
Bolt confidence: 88% W: 69.7px, H: 65.6px
Bolt confidence: 76% W: 27.8px, H: 29.3px
Anchor confidence: 76% W: 163.7px, H: 88.0px
Anchor confidence: 53% W: 105.9px, H: 210.5px
Anchor confidence: 88% W: 119.2px, H: 109.9px
Anchor confidence: 78% W: 156.4px, H: 225.9px

Figure 124: True Positives for Combined Model

Also anchors are detected with high confidence levels. The true positives for anchors are shown in the following Figure 125.

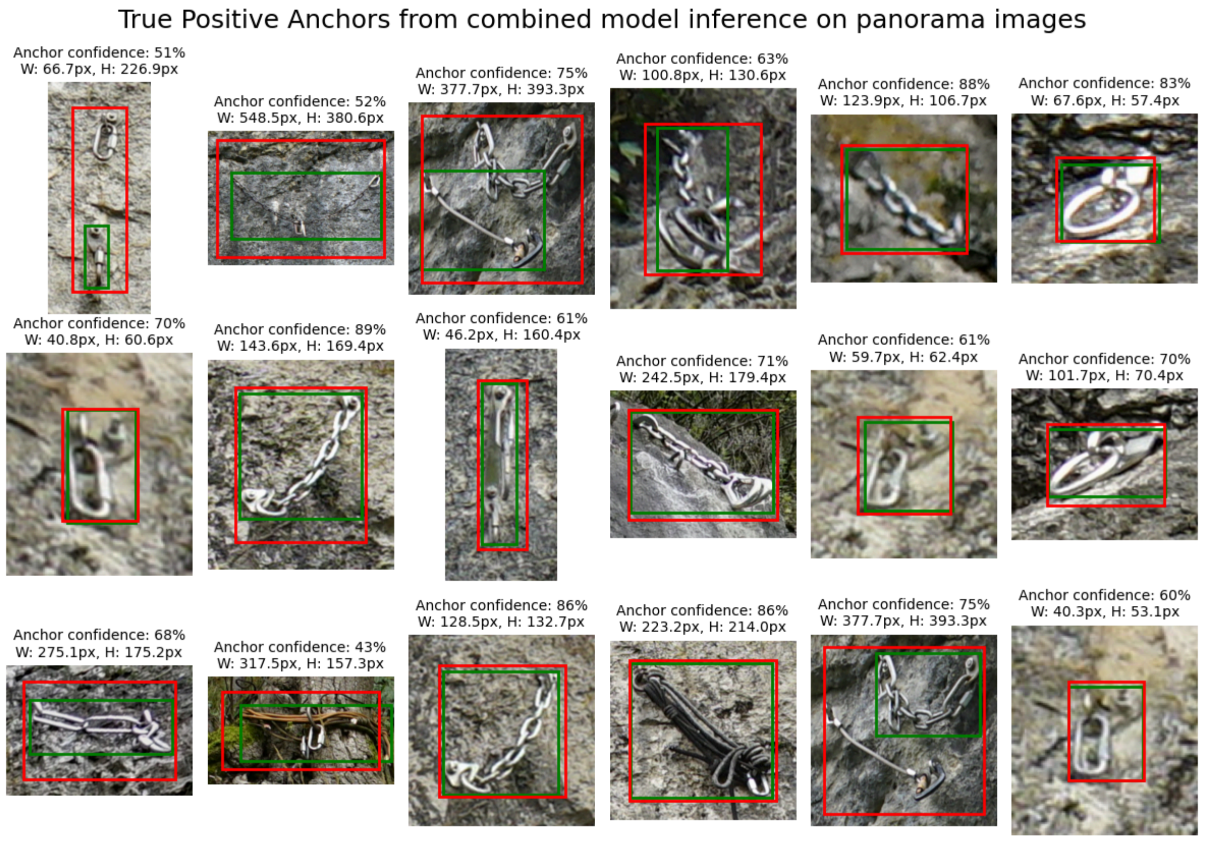

True Positive Anchors from combined model inference on panorama images
Anchor confidence: 51% W: 66.7px, H: 226.9px
Anchor confidence: 52% W: 548.5px, H: 380.6px
Anchor confidence: 75% W: 377.7px, H: 393.3px
Anchor confidence: 63% W: 100.8px, H: 130.6px
Anchor confidence: 88% W: 123.9px, H: 106.7px
Anchor confidence: 83% W: 67.6px, H: 57.4px
Anchor confidence: 70% W: 40.8px, H: 60.6px
Anchor confidence: 89% W: 143.6px, H: 169.4px
Anchor confidence: 61% W: 46.2px, H: 160.4px
Anchor confidence: 71% W: 242.5px, H: 179.4px
Anchor confidence: 61% W: 59.7px, H: 62.4px
Anchor confidence: 70% W: 101.7px, H: 70.4px
Anchor confidence: 68% W: 275.1px, H: 175.2px
Anchor confidence: 43% W: 317.5px, H: 157.3px
Anchor confidence: 86% W: 128.5px, H: 132.7px
Anchor confidence: 86% W: 223.2px, H: 214.0px
Anchor confidence: 75% W: 377.7px, H: 393.3px
Anchor confidence: 60% W: 40.3px, H: 53.1px

Figure 125: True Positives for Combined Model

II 6.1.1.4.5 False Positives and False Negatives

Looking at the false positives and false negatives of the combined model in the following Figure 127 and Figure 126, gives an understanding of what the model is still struggling with.

The model is struggling with:
- vegetation like branches.
- obscured bolts and anchors.
- low resolution details.
- highlights on objects.

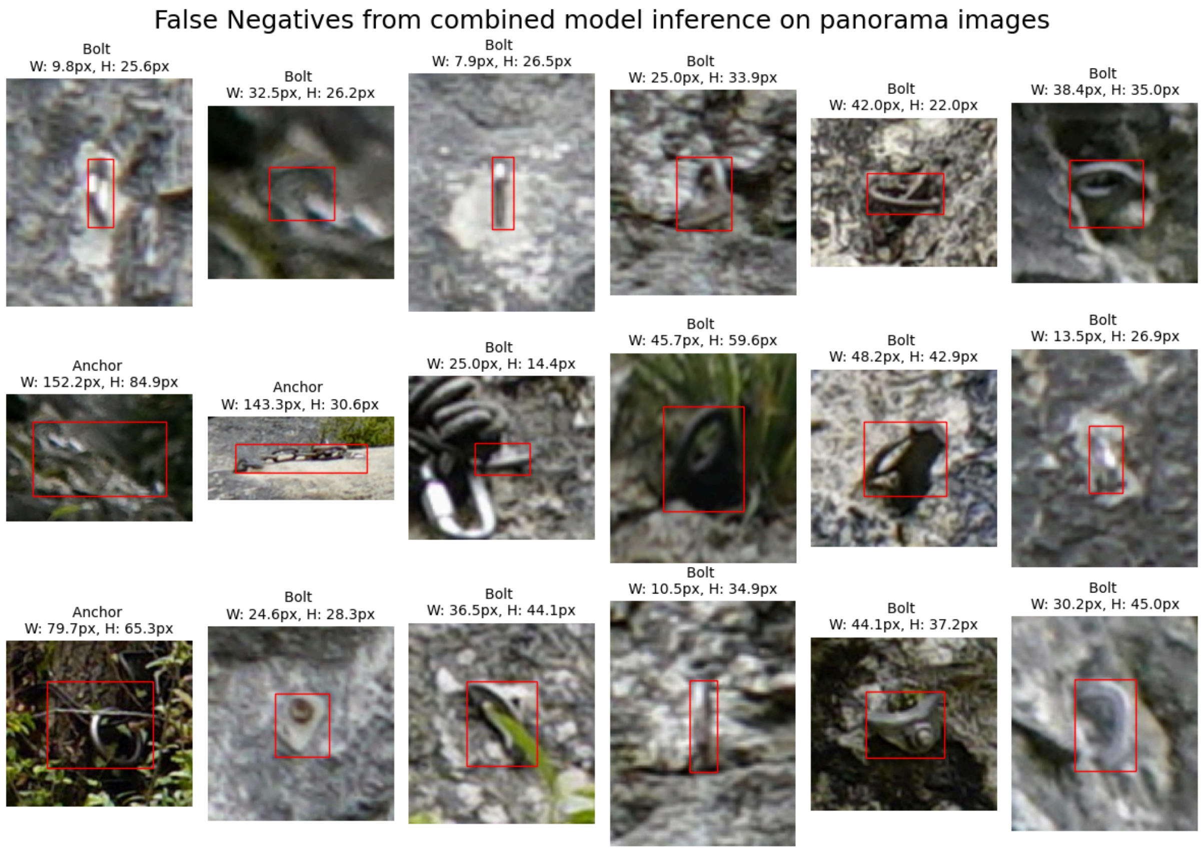

False Negatives from combined model inference on panorama images
Bolt W: 9.8px, H: 25.6px
Bolt W: 32.5px, H: 26.2px
Bolt W: 7.9px, H: 26.5px
Bolt W: 25.0px, H: 33.9px
Bolt W: 42.0px, H: 22.0px
Bolt W: 38.4px, H: 35.0px
Anchor W: 152.2px, H: 84.9px
Anchor W: 143.3px, H: 30.6px
Bolt W: 25.0px, H: 14.4px
Bolt W: 45.7px, H: 59.6px
Bolt W: 48.2px, H: 42.9px
Bolt W: 13.5px, H: 26.9px
Anchor W: 79.7px, H: 65.3px
Bolt W: 24.6px, H: 28.3px
Bolt W: 36.5px, H: 44.1px
Bolt W: 10.5px, H: 34.9px
Bolt W: 44.1px, H: 37.2px
Bolt W: 30.2px, H: 45.0px

Figure 126: False Negatives for Combined Model

These are some of the false positives of the combined model. The Figure 127 also shows the confidence levels of the false positives.

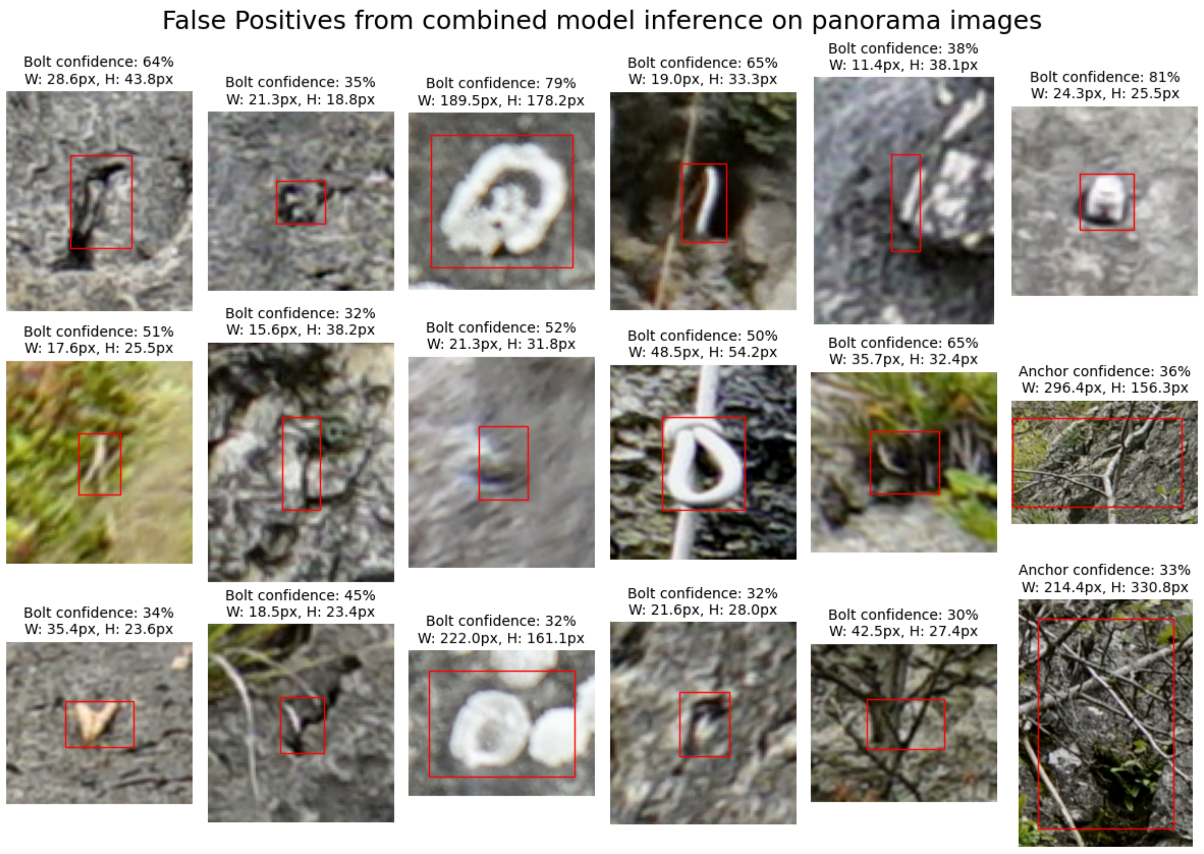


Figure 127: False Positives for Combined Model

II 6.1.2 Results of the Algorithm

The route detection algorithm was evaluated using a comprehensive testing setup designed to measure its performance and identify areas for improvement. The evaluation focuses on predicting climbing routes from bolts and anchors in panoramic images and analyzing the results quantitatively and visually.

II 6.1.2.1 Testing Methodology

The algorithm generates routes from bolts and anchors, and the results are evaluated against expected routes defined in a series of test cases. Custom evaluation functions check the correctness of the predicted routes, and visual overlays of the routes on test images are used to identify discrepancies.

The evaluation criteria categorize routes as follows:

- **Correct**: All routes are correctly generated, with no missing or extra bolts.
- **Minor Errors**: Routes contain up to one mistake, such as a missing or extra bolt.
- **Major Errors**: Routes contain more than one mistake, such as multiple missing or extra bolts or entirely incorrect connections.

II 6.1.2.2 Performance Metrics

The following Table 32 summarizes the results for individual test images:

Image	Total Routes	Correct Routes	Minor Errors	Major Errors
02_center_2_cropped_left	17	15	2	0
02_center_2_cropped	18	15	3	0
03_ontheleft	16	13	3	0
04_right	16	11	5	0
20_middle_left	22	16	6	0
11_left	13	10	2	1

Table 32: Performance Metrics: Per Image Results

Overall, the algorithm achieves the following results across all test cases:

- **Total Routes Tested**: 102 routes.
- **Correct Routes**: 80 routes (78%).
- **Minor Errors**: 21 routes (21%).
- **Major Errors**: 1 route (1%).

The Figure 128 visualization provides a detailed overview of the algorithm's performance:

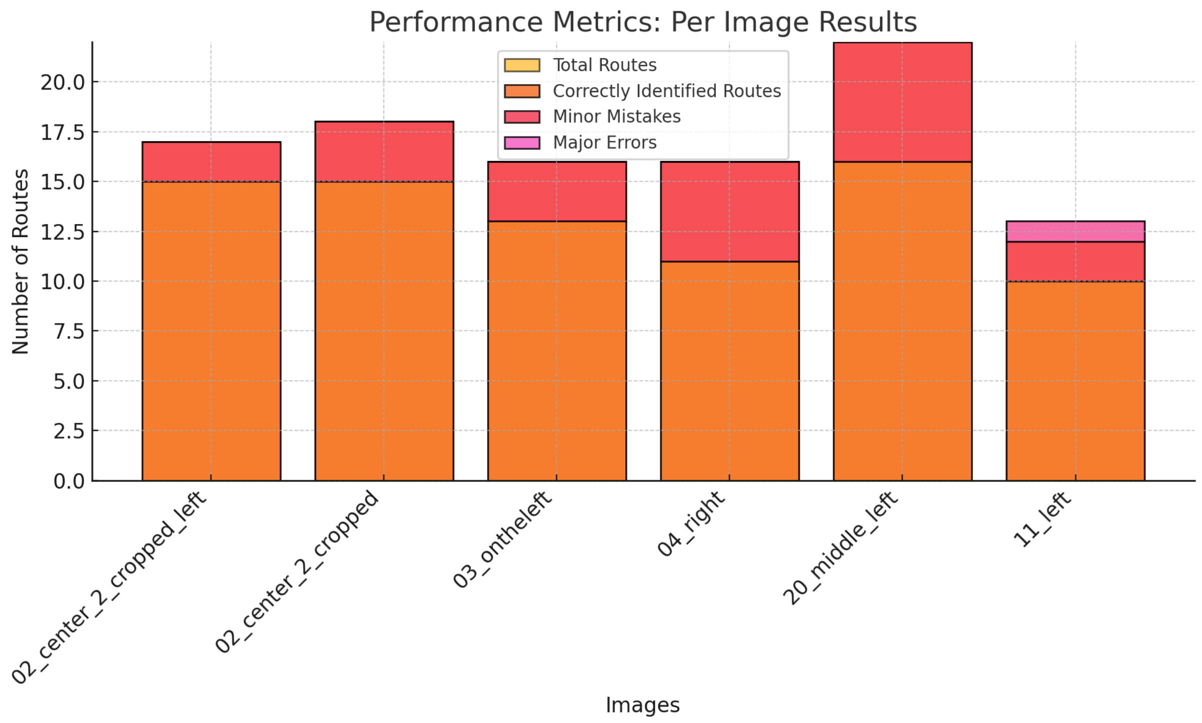


Figure 128: Overall Performance Metrics.

These results highlight the algorithm's high accuracy, with the majority of routes being correctly generated. Minor errors typically occur in complex cases where bolts are closely clustered, while major errors are rare.

II 6.1.2.3 Orientation-Based Limitations

The algorithm's performance is influenced by the perspective of the climbing wall, as shown in Figure 129. The wall is divided into three sections based on orientation:

- **Green Section**: Forward-facing walls. Routes in this section are consistently correct.
- **Yellow Section**: Slightly angled walls. Routes in this section often contain minor errors, such as missing or extra bolts.
- **Red Section**: Steep walls facing away. Routes in this section frequently encounter major errors, with incorrect starting points or disconnected bolts.

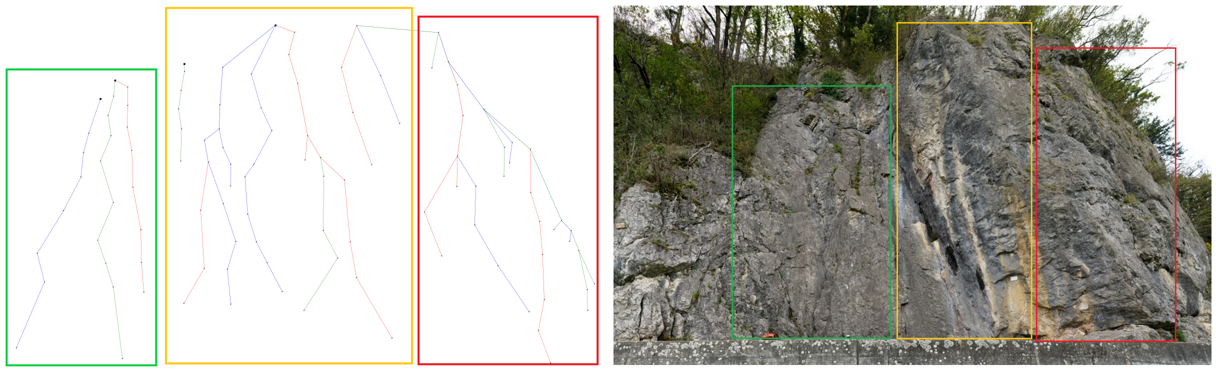


Figure 129: Route detection accuracy across wall orientations. Left: Generated climbing routes. Right: Actual wall with sections highlighted in green (front-facing), yellow (angled), and red (steep).

II 6.1.2.4 Conclusion

The evaluation confirms the algorithm's ability to detect and map climbing routes with a high degree of accuracy, particularly on forward-facing walls. While minor errors are manageable and mostly occur in clustered configurations, major errors highlight the need for improved handling of challenging perspectives and overlapping predictions. By addressing these limitations, the algorithm can achieve greater robustness and precision in future iterations.

II 6.1.3 Results: Requirements Fulfillment

Both functional and non-functional requirements are evaluated in this section to determine the system's adherence to the defined expectations.

II 6.1.3.1 Functional Requirements (FR)

The following Table 33 summarizes the fulfillment of the defined functional requirements:

FR	Description	Fulfillment Status
FR1	Detect climbing bolts in images with F1 score >= 80% and recall >= 90%.	Fulfilled (F1: 82%, Recall: 92%).f1: Figure 121 recall: Figure 122
FR2	Generate route structure from detected bolts with <30% minor errors and <10% major errors.	Fulfilled (Major errors: 1%). Chapter II 6.1.2.2
FR3	Allow manual adjustment of detected bolts and anchors.	Fulfilled (User test Chapter II 5.10).
FR4	Support adding new bolts or anchors.	Fulfilled (User test Chapter II 5.10).
FR5	Support manual modification of routes.	Fulfilled (User test Chapter II 5.10).
FR6	Support deletion or merging of routes.	Fulfilled (User test Chapter II 5.10).
FR7	Provide visual mapping of computed routes.	Fulfilled (User test Chapter II 5.10).
FR8	Deliver a user-friendly interface for all functionalities.	Not Fulfilled (No user study conducted due to time restrictions, as the focus of the project is not on user-friendliness).

Table 33: Functional Requirements Fulfillment

II 6.1.3.2 Non-Functional Requirements (NFR)

The following Table 34 showcases the status of non-functional requirements:

NFR	Description	Fulfillment Status
NFR1	Time Behavior: Response time <= 2s for 90% of processes.	Fulfilled (User test Chapter II 5.10: Actions taking longer than 2s provide clear status updates to inform the user of ongoing processes).
NFR2	User Error Protection: Confirmations for destructive actions.	Partially Fulfilled (User test Chapter II 5.10: Confirmation modal present for recalculating routes, but missing for deleting a job).
NFR3	Modularity: Independent component updates.	Fulfilled (Figure 19).
NFR4	Compatibility: Endpoints return defined data structures.	Fulfilled (Chapter III 5).

Table 34: Non Functional Requirements Fulfillment

II 6.1.4 Conclusion

Out of the 8 defined Functional Requirements (FR), 7 are fulfilled, while 1 is not fulfilled due to the lack of a user study focused on user-friendliness. For the Non-Functional Requirements (NFR), 3 are fully fulfilled, and 1 is partially fulfilled, demonstrating the system's strong adherence to most functional and non-functional expectations, with room for improvement in destructive action confirmation and user interface evaluation.

II 6.2 Further Development

The next steps and possible further developments are described in Chapter I 5.3.

II 7 Software Operation Documentation

This chapter explains how to setup and use the software developed in this project.

II 7.1 Requirements

The components of the software have different hardware and software requirements in order to run. Most of the dependencies are installed via Docker so the installation process is simplified.

II 7.1.1 Dockerized Software Components:

For the dockerized software components shown in Chapter II 5.6 the following requirements are needed:

- Host machine with Docker & Docker Compose installed
  - mi. 32GB RAM
  - mi. 8 CPU cores
  - mi. 200GB disk space
  - mi. 1 GPU with CUDA support
    - min. 8GB VRAM
    - (for example NVIDIA RTX 4060 or better)

II 7.1.2 ML Model Training:

Dataset creation and model training can be done on the same machine as the dockerized components run on but it should have all it's resources available for the training process.

The training step can be executed in three ways:

- Over a Jupyter Notebook
- Over a Docker container (which is running a python script)
- Over a Apptainer running a Jupyter Notebook (usable on the OST DGX server)

For the model training and testing the following host options are recommended:

1. Host machine
   - mi. 32GB RAM
   - mi. 8 CPU cores
   - mi. 200GB disk space
   - mi. 1 GPU with CUDA support
     - min. 8GB VRAM
     - (for example NVIDIA RTX 4060 or better)
2. OST DGX Server
   - 32GB RAM
   - 80 CPU cores
   - 500GB disk space
   - 2 GPU's with CUDA support
     - 16GB VRAM
     - (for example NVIDIA Tesla P100-SXM2-16GB or better)

II 7.2 Installation

Follow the readme.md files in the respective repositories to install the software.

II 7.3 Guide for Getting Dataset

This guide explains how to capture images of climbing walls with bolts and anchors for training the machine learning model or for image format to find routes. The images are stitched together to create a panorama, which is then labeled using the LabelMe tool.

A pdf version of this guide can be found in the model repository under `./dataset_tutorial.pdf`.

II 7.3.1 Steps

1. **Organize a Camera with Specifications Similar to These**:
   - 35mm full-frame sensor
   - 42.4 megapixels
   - Saves images as JPEG in the highest quality
   - Use a 75mm lens or higher to take images from a distance.
2. **Position the Camera**
   - Place it on a tripod about 8 meters away from the wall (around the distance where bolts are still visible to the eye).
   - Ensure the bolts are approximately 20-60px (preferably 60px) in height or width.
   - Make sure the lighting is even across the wall. Take the image with no high contrast shadows.
   - Take pictures of the wall by dividing it into a grid and capturing each cell with enough **overlap** to facilitate stitching. Generally, 3 to 6 vertical shots should be enough, but follow a consistent pattern, like shown in Figure 130:

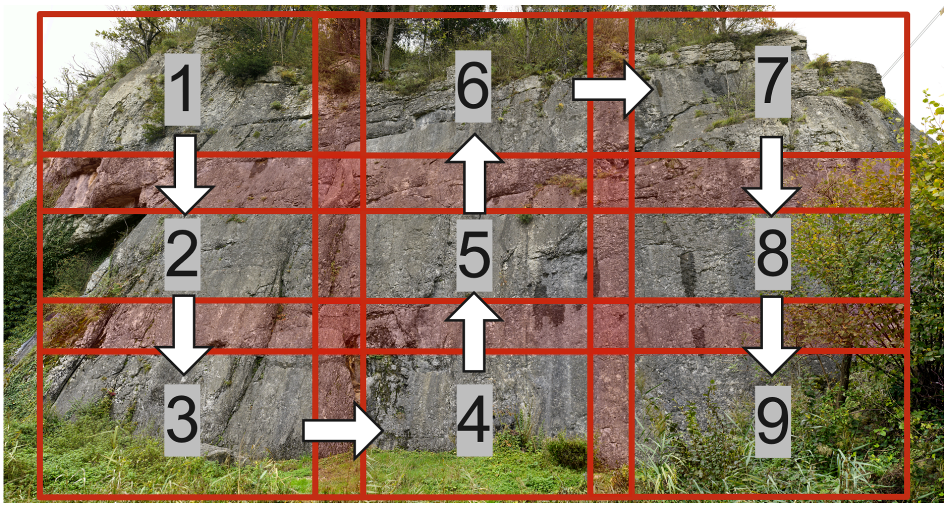


Figure 130: Guide for stitching images

   - **Tip:** For each panorama, create a new folder on the camera to simplify organizing images later.
3. **Stitch the Images**

   Use the Image Composite Editor:
   https://www.microsoft.com/en-us/research/project/image-composite-editor/
   - Download via: https://www.chip.de/downloads/Microsoft-Image-Composite-Editor-64-Bit_42077502.html
   - Select the images to stitch and start the process (Figure 131):

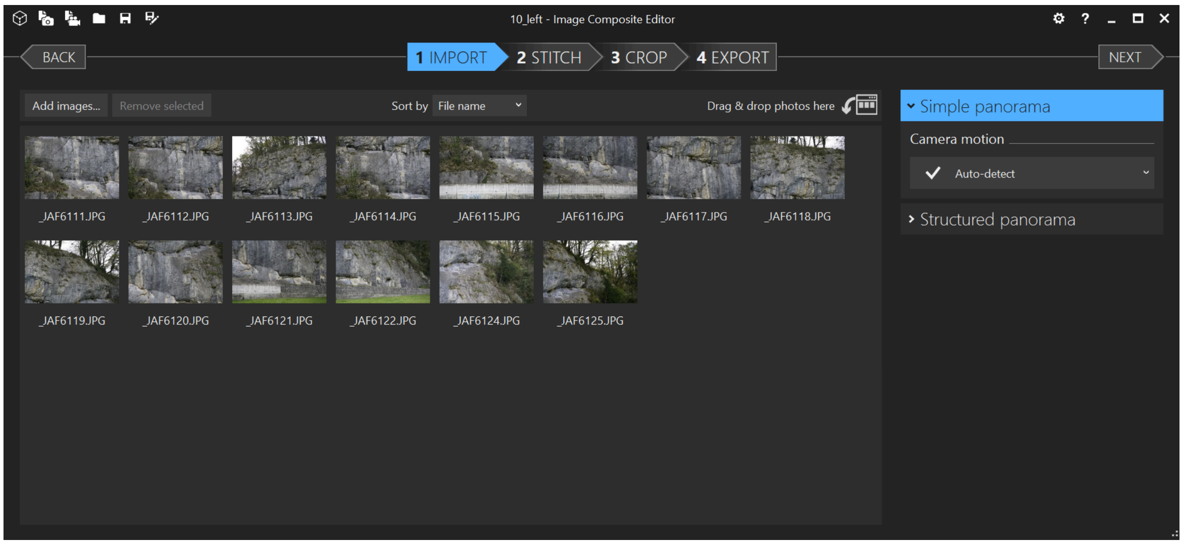


Figure 131: Image Composite Editor (ICE) interface

- Choose the perspective projection.
- Adjust the alignment by centering, moving, or rotating the image as needed.
- Crop or generate content for empty spaces in the image.
- Export it as a 100% quality JPEG in Ultra resolution.

4. **Label the Stitched Image**
   - Download and install LabelMe:
     https://www.labelme.io/install or https://github.com/wkentaro/labelme
   - Open the folder with the images (loading may take time since stitched images are 300–800 MB).

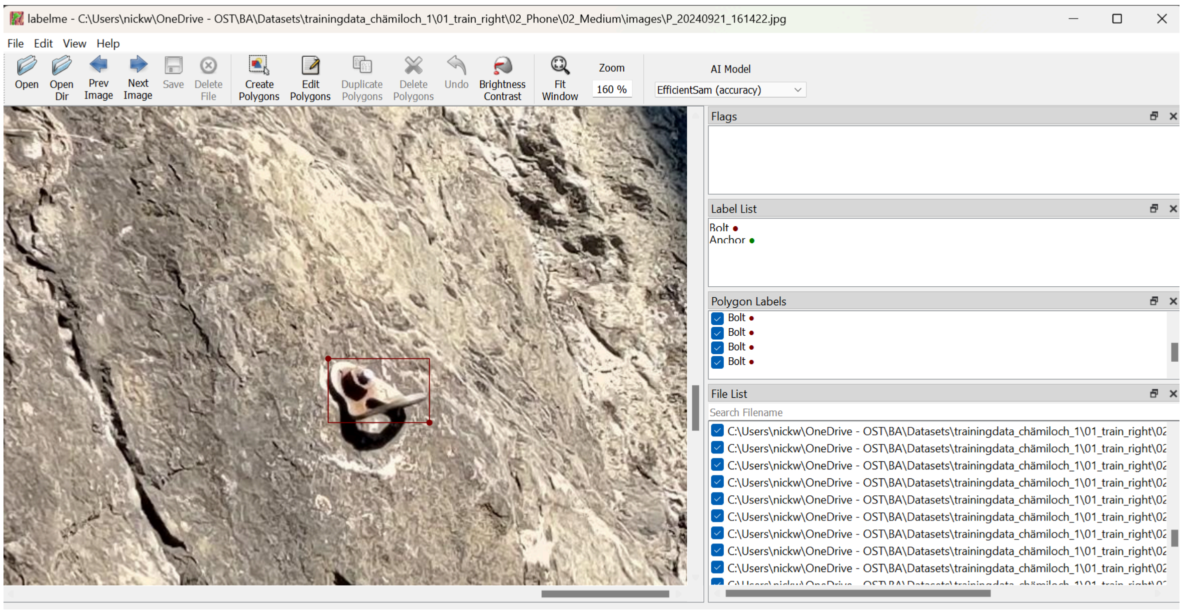


Figure 132: LabelMe UI

- Open an image and start labeling the bolts (Figure 132):
  - Zoom in and select "Create Rectangle" via right-click.
  - Select two points to create a rectangle around each bolt or anchor.
  - Use the labels "Bolt" and "Anchor."
  - Ensure the rectangle closely fits around each bolt or anchor with minimal margin.
  - Press **CTRL+S** to save the annotation file in the same folder.

III Appendix A

III 1 Task Definition

This chapter was created at the beginning of the project to define the tasks and goals of the project.

III 1.1 Project Title

BoltFinder - Pilot project with SAC for automated climbing route inventory keeping

III 1.2 Persons Involved

Supervisors
- Mitra Purandare
- Lars Herrmann

Students
- Nick Wallner
- Fabio Elvedi

Industry partner
- Swiss Alpine-Club SAC
  Marek Polacek

III 1.3 Aim of the Thesis

The aim of this bachelor's thesis is to design and develop a proof of concept (POC) for the digitization of climbing routes in Switzerland. The routes are to be captured automatically on the basis of data (e.g. photo series, drone recordings, GPS route course). To this end, an object recognition model is to be trained specifically on climbing infrastructure such as bolts and rope rings in order to identify their exact position and possibly their condition. The recognized nodes on the rock are then transformed into a data structure, e.g. in the form of a directed graph. Possibilities for automatically determining possible routes from the data obtained will also be investigated. A further aim is to create a basis for visualizing the data in various formats such as sketches, 3D models and topographies. The work aims to develop and evaluate an innovative proof-of-concept solution for recording and managing climbing routes in Switzerland in order to demonstrate the potential for future optimization of the SAC tour portal.

III 1.4 Tasks

The following tasks are to be completed as part of this thesis:

III 1.4.1 Analysis:

- Identification of functional requirements (FR) for capturing and displaying climbing routes.
- Investigation of non-functional requirements (NFR), such as quality, reliability, and integration into the SAC tour portal.
- Analysis of existing methods for recording routes.

III 1.4.2 Architecture & Design:

- Selection of an appropriate architecture for the machine-learning model for object detection and route analysis.
- Investigation and selection of technologies for the project.
- Examination of possibilities for combining multiple measurement points (e.g., images) for joint processing in a model.
- Design of a data model for storing and subsequently processing the captured route information.
- Examination of possibilities for visualizing the collected data.

III 1.4.3 Implementation & Testing:

- Implementation of functional requirements (FR) for route capture and display.
- Development and training of an object recognition model on climbing infrastructure.
- Documentation of results and potential extensions for future applications.

III 1.5 Expected Results:

- A functioning proof-of-concept for automated route capture.
- Automated generation of visualizations of the discovered routes.
- Evaluation of the solution and recommendations for future improvements.

III 1.6 Deadlines

- 16.09.2024, Start of the thesis.
- 10.01.2025, Submission of the thesis.

III 1.7 Assessment

The supervisors are responsible for the evaluation. The weighting of the assessment follows the guidelines for bachelor/master theses.

Criterion	Weight
Organization and Execution	10%
Formal Quality of the Report	10%
Analysis, Design, and Evaluation	20%
Technical Implementation	40%
Bachelor examination	20%

Table 35: Functional Requirements Fulfillment

III 1.8 Permitted Tools and Additional Support

If necessary, assistance in the areas of image processing and machine learning will be provided.

III 2 Mockups

This section provides detailed mockups of the user interface design, including simple and advanced workflows.

III 2.1 Simple Workflow

The simple workflow covers:

- Uploading images of climbing walls.
- Generating routes.
- Exporting topographic maps.

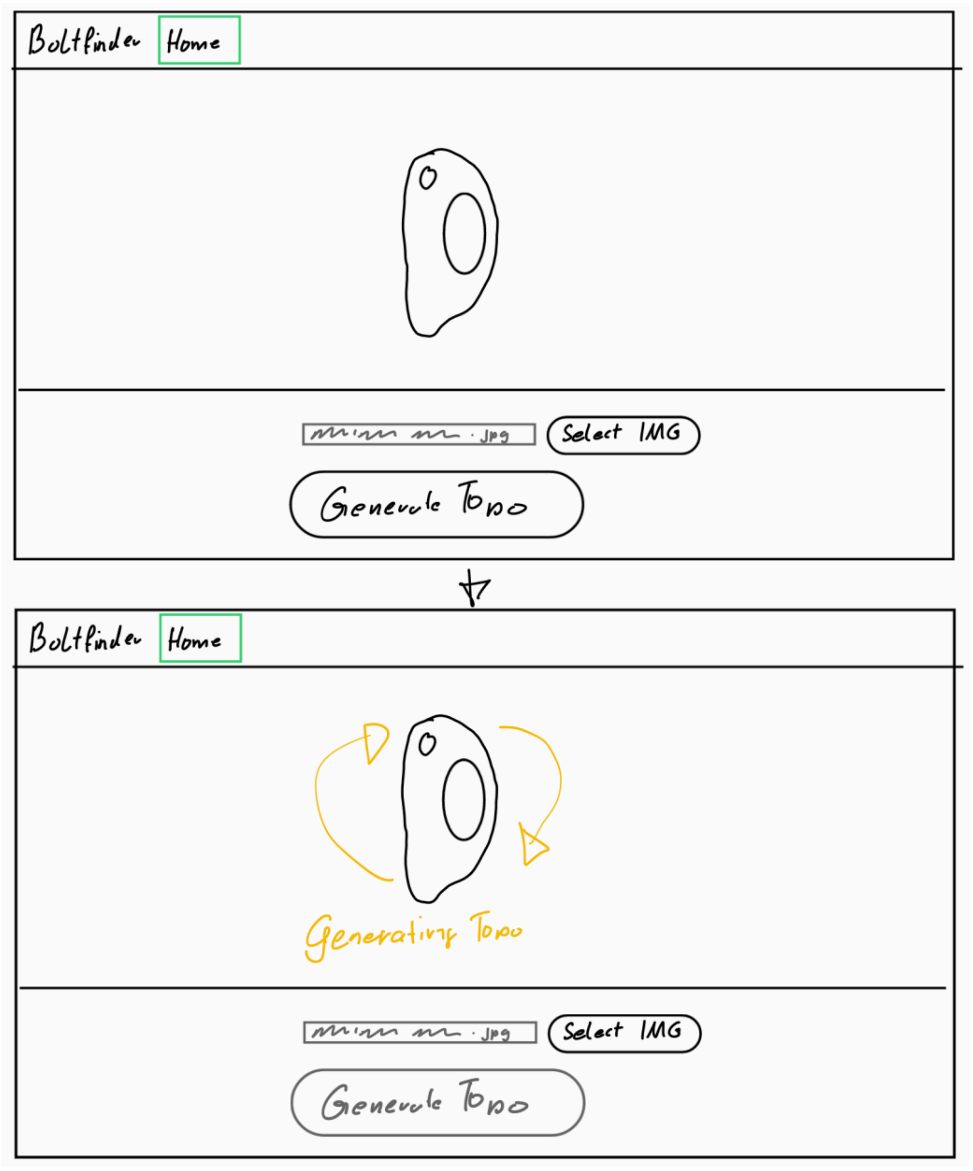


Figure 133: Home page and generator screen.

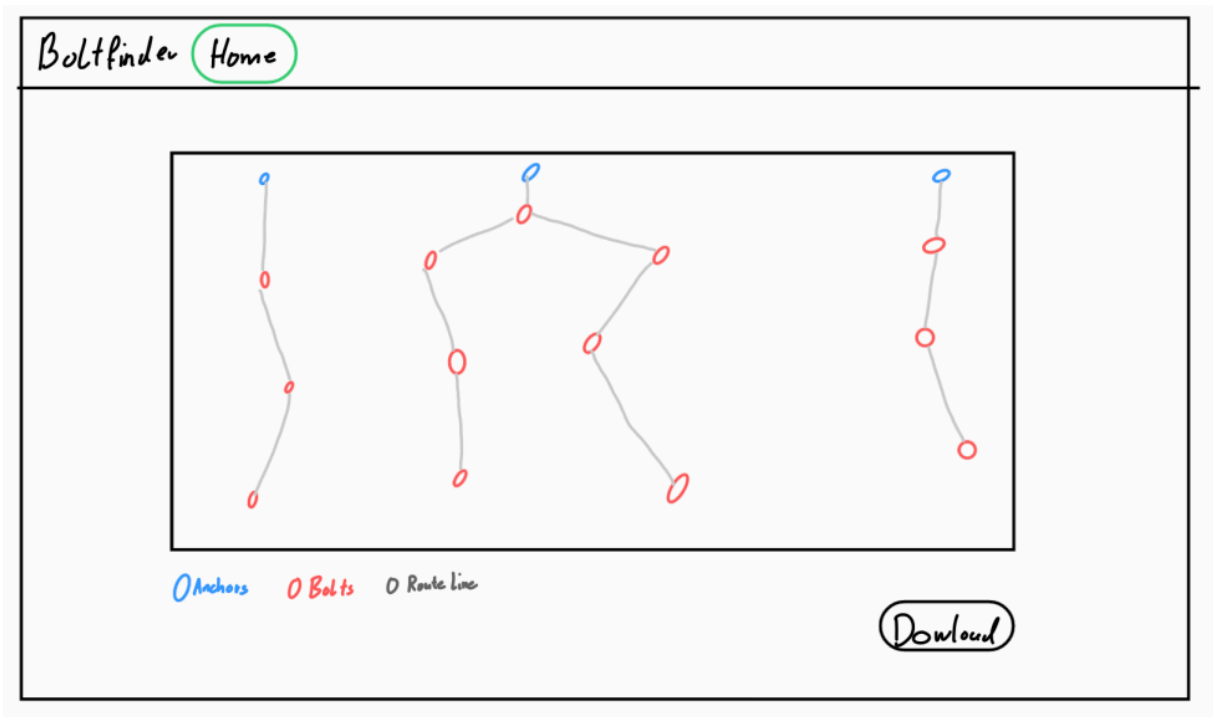


Figure 134: Route detection and export screen.

III 2.2 Advanced Workflow

The advanced workflow introduces:

- Editing bolts and anchors.
- Editing climbing routes.
- Viewing the history of route edits.

Its shown in the following images.

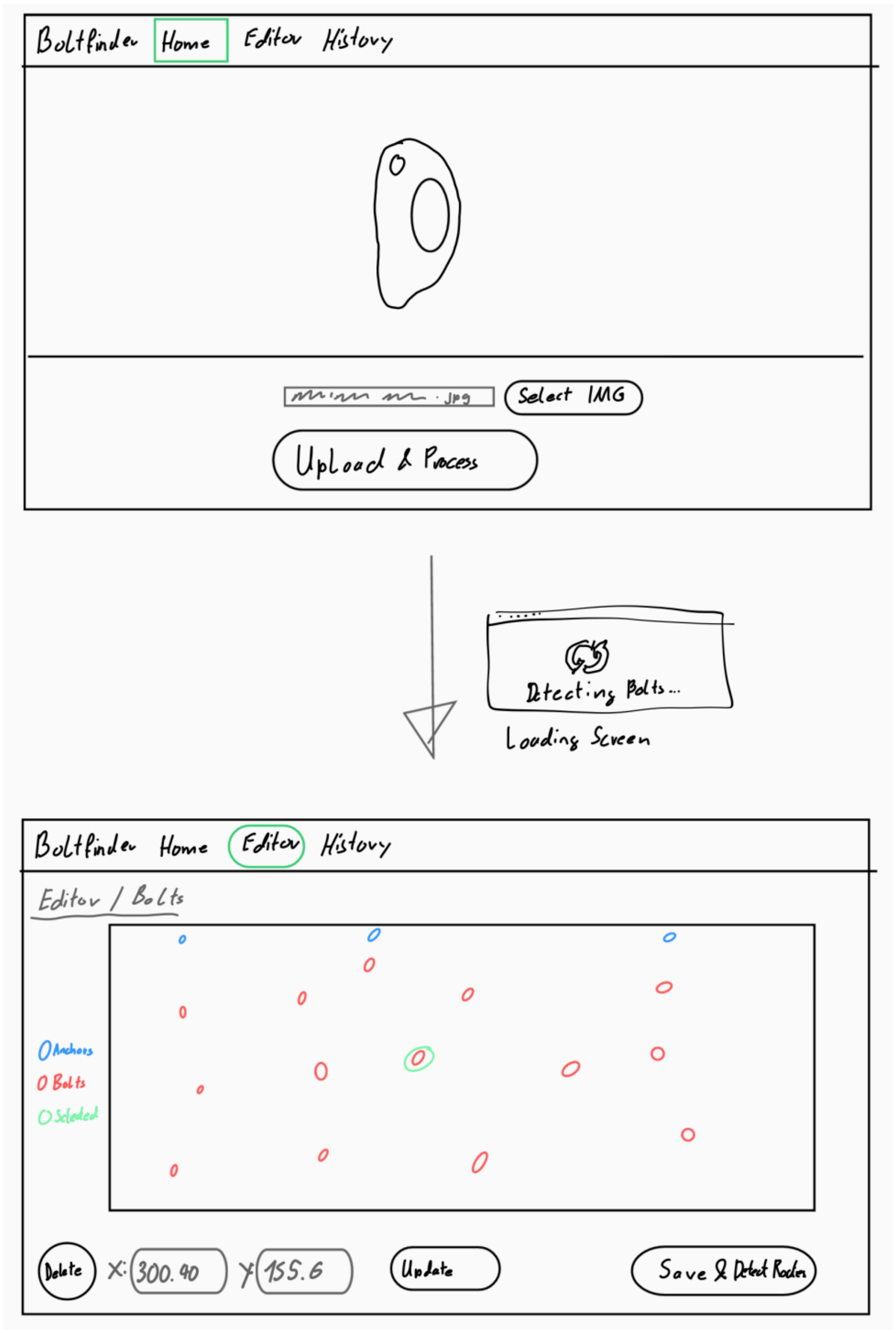
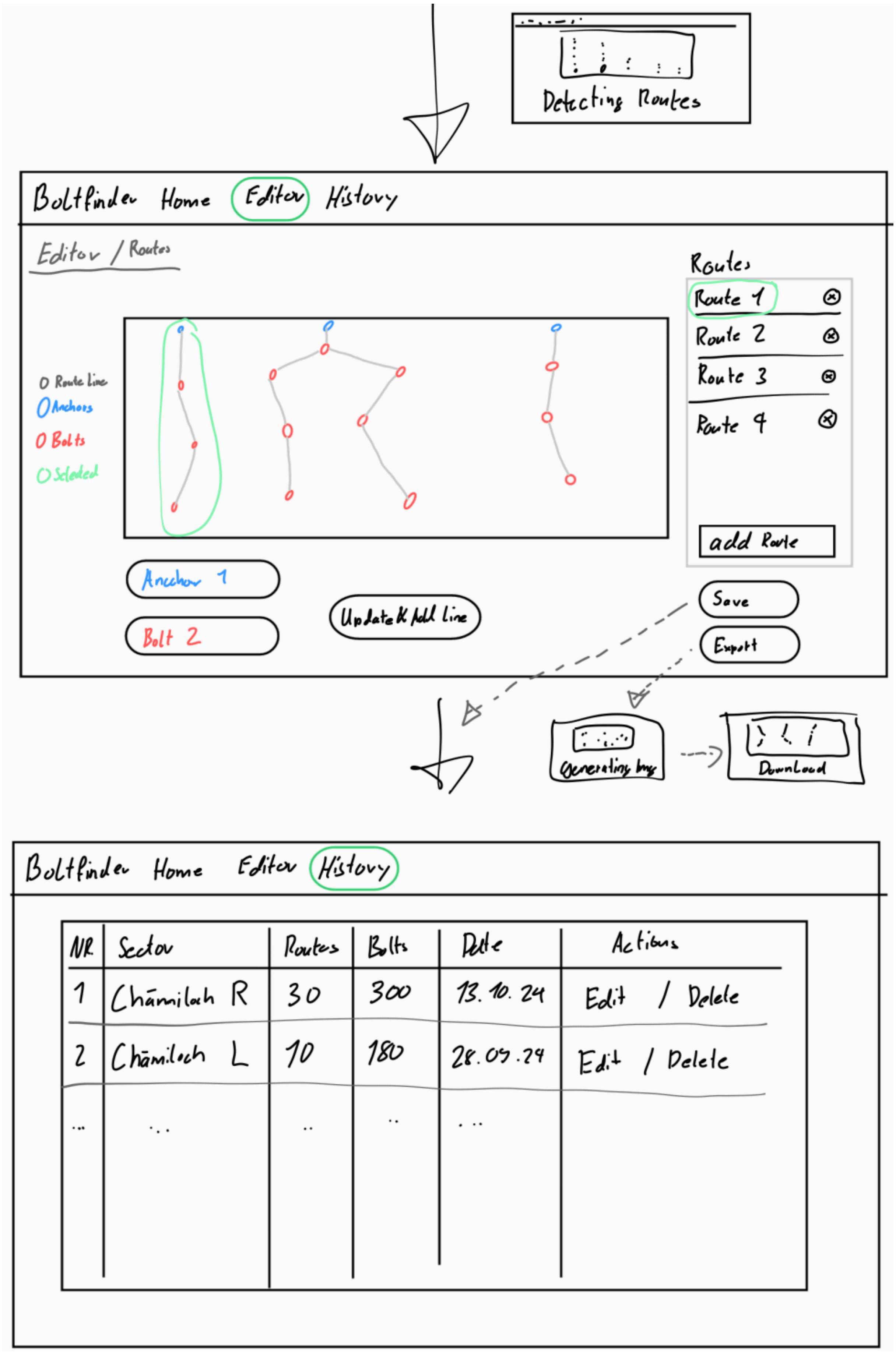


Figure 135: Home page and bolt/anchor editor screen.



NR	Sector	Routes	Bolts	Date	Actions
1	Chämilach R	30	300	13. 10. 24	Edit / Delete
2	Chämilach L	10	180	28. 09. 24	Edit / Delete
..	...	..	..	...	

Figure 136: Route editor, export, and history screen.

III 3 Further Model Results

Here are more of the results listed, which are not included in the main documentation.

III 3.1 Faster R-CNN vs YOLO Model Comparison

By training and evaluating multiple versions of the "Faster R-CNN" and "YOLOv11" models, the YOLO model proves to be the better fitting model for detecting bolts and anchors in the high resolution panorama images. Figure 137

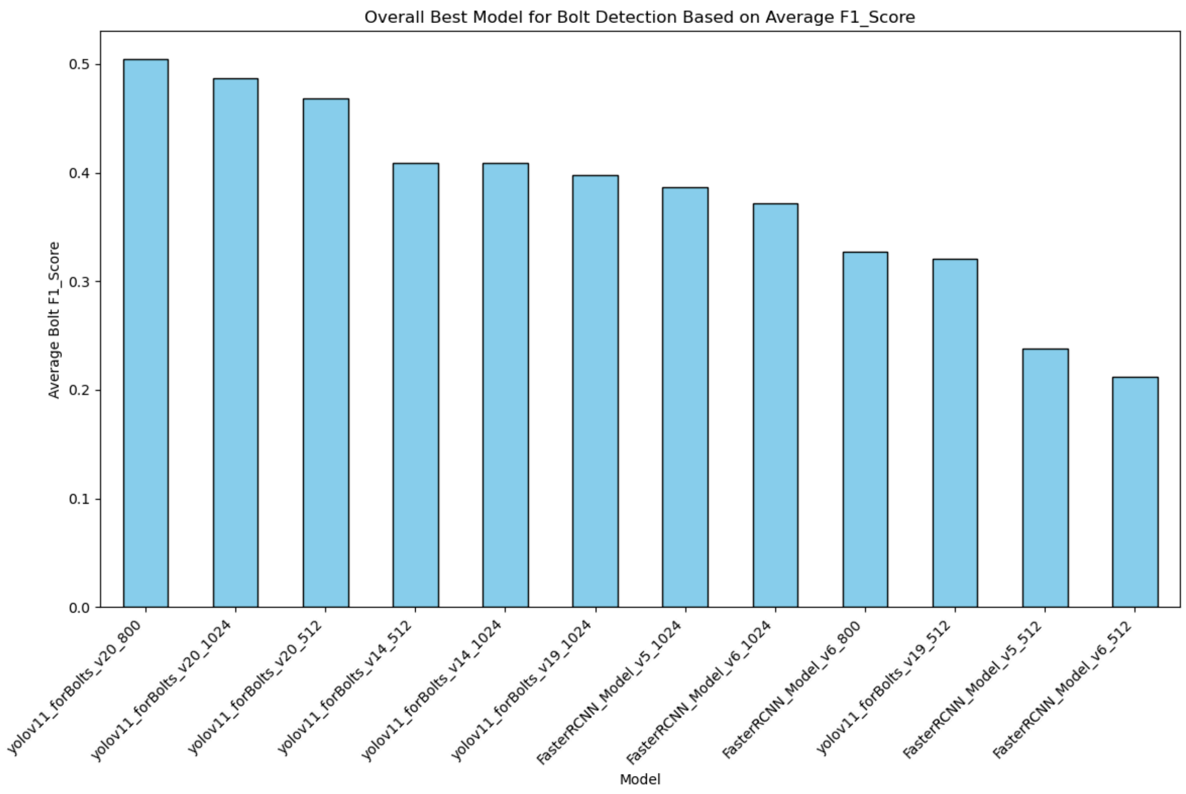


Figure 137: Overall best bolt detection based on average F1 score

Models are trained and evaluated with different tile sizes (1024, 800, 512) and different parts of the dataset.

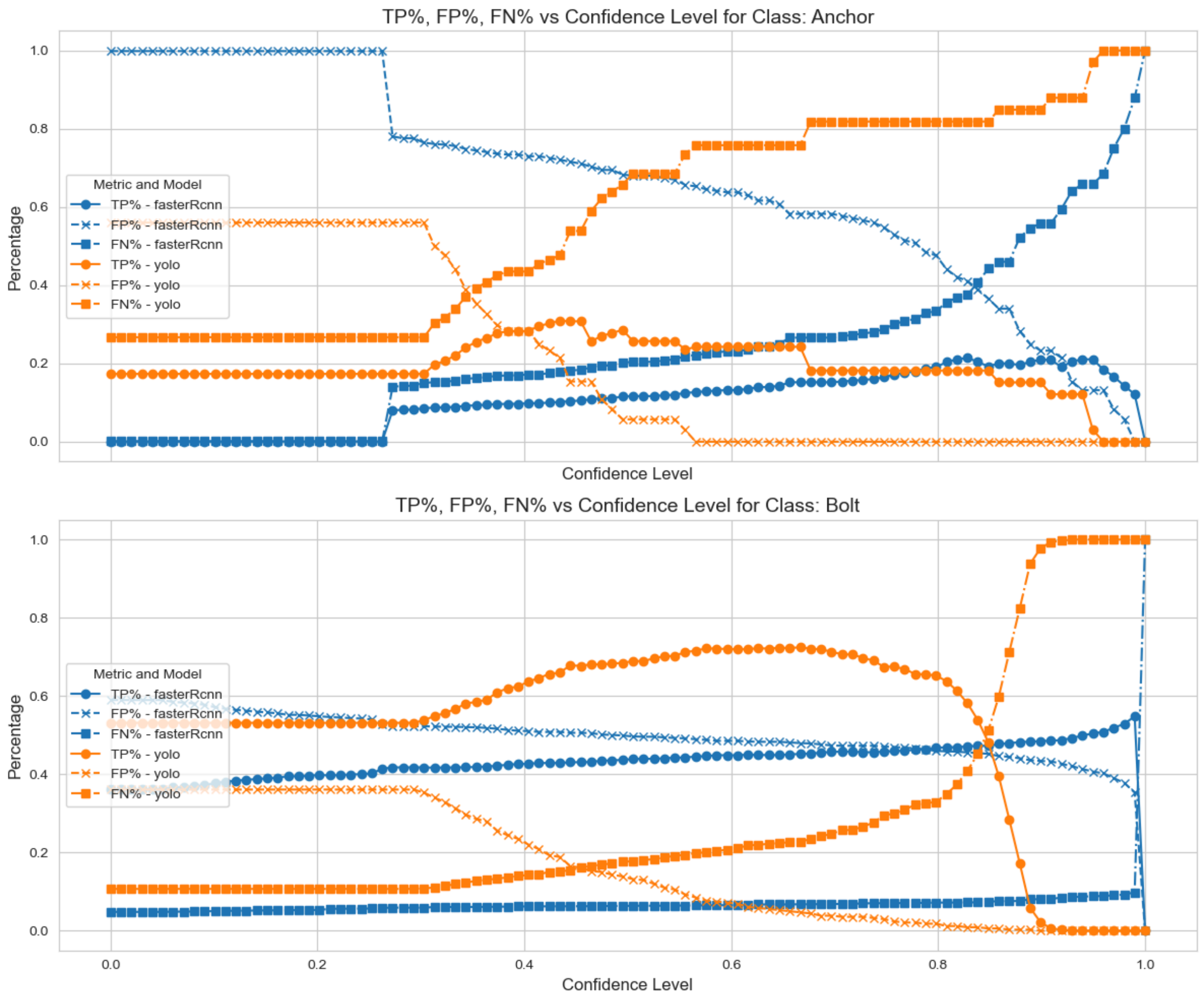


Figure 138: Faster R-CNN vs YOLOv11 Comparison True Positives, False Negatives and False Positives in percentages

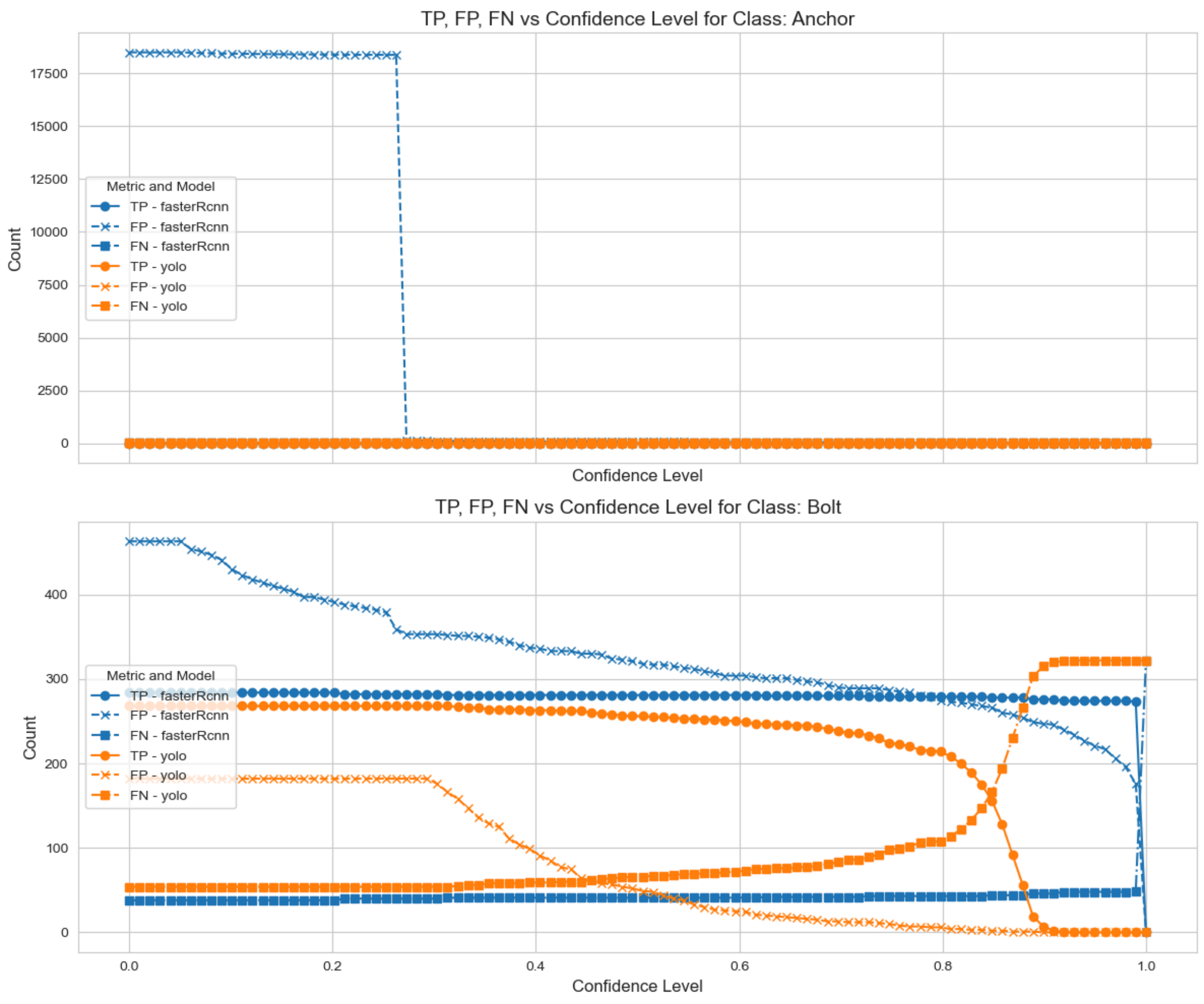


Figure 139: Faster R-CNN vs YOLOv11 Comparison True Positives, False Negatives and False Positives in absolute numbers

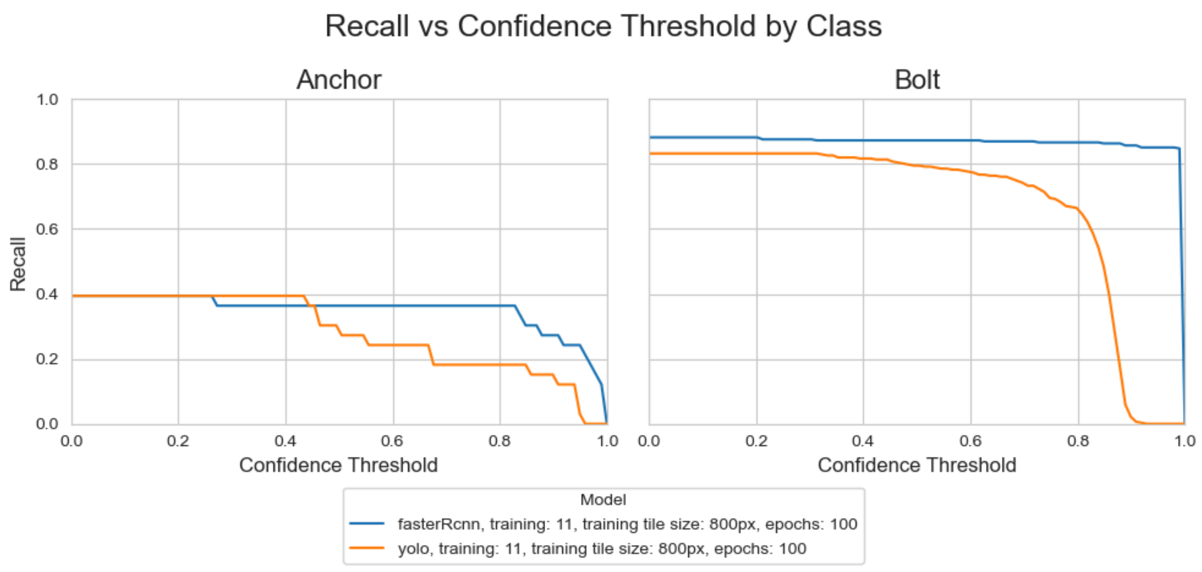


Figure 140: Faster R-CNN vs YOLOv11 Recall Comparison

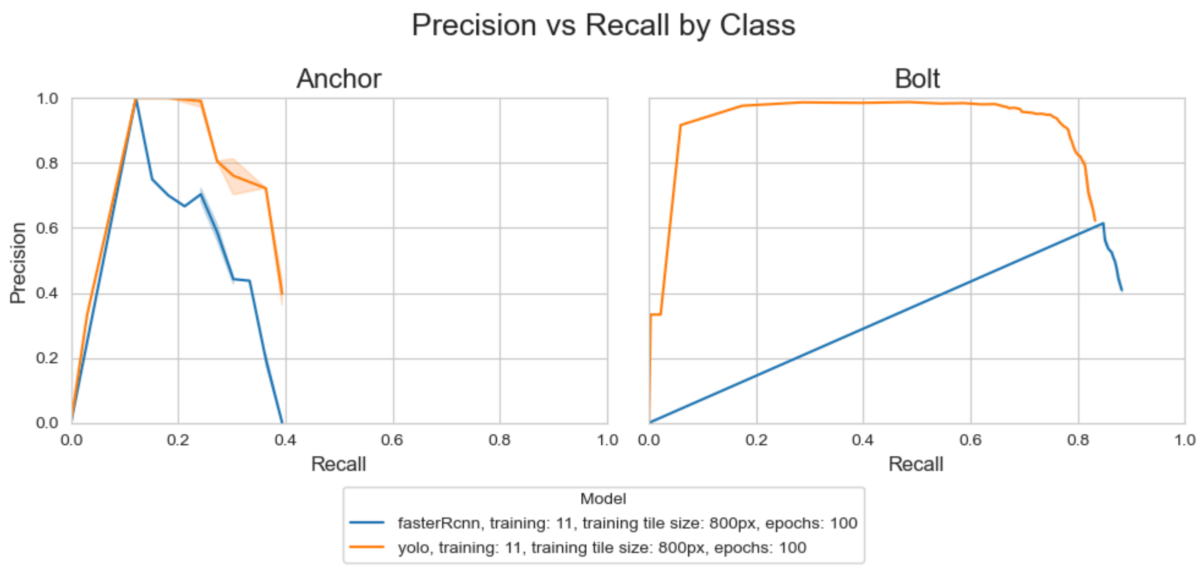


Figure 141: Faster R-CNN vs YOLOv11 Precision Recall Comparison

III 3.2 Tile Size Comparison by Class

III 3.2.1 Mean Average Precision

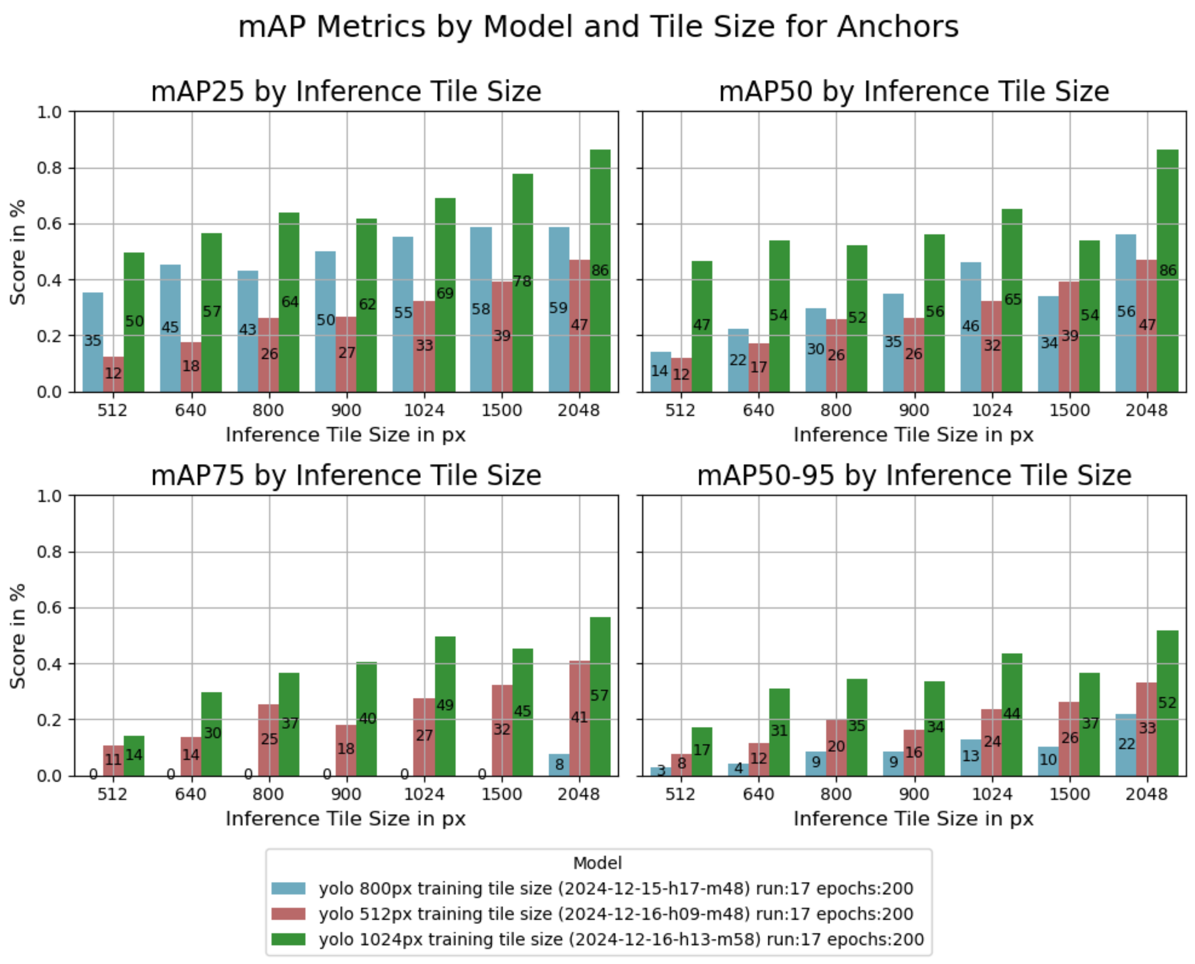


Figure 142: Tile Size Comparison for Anchors

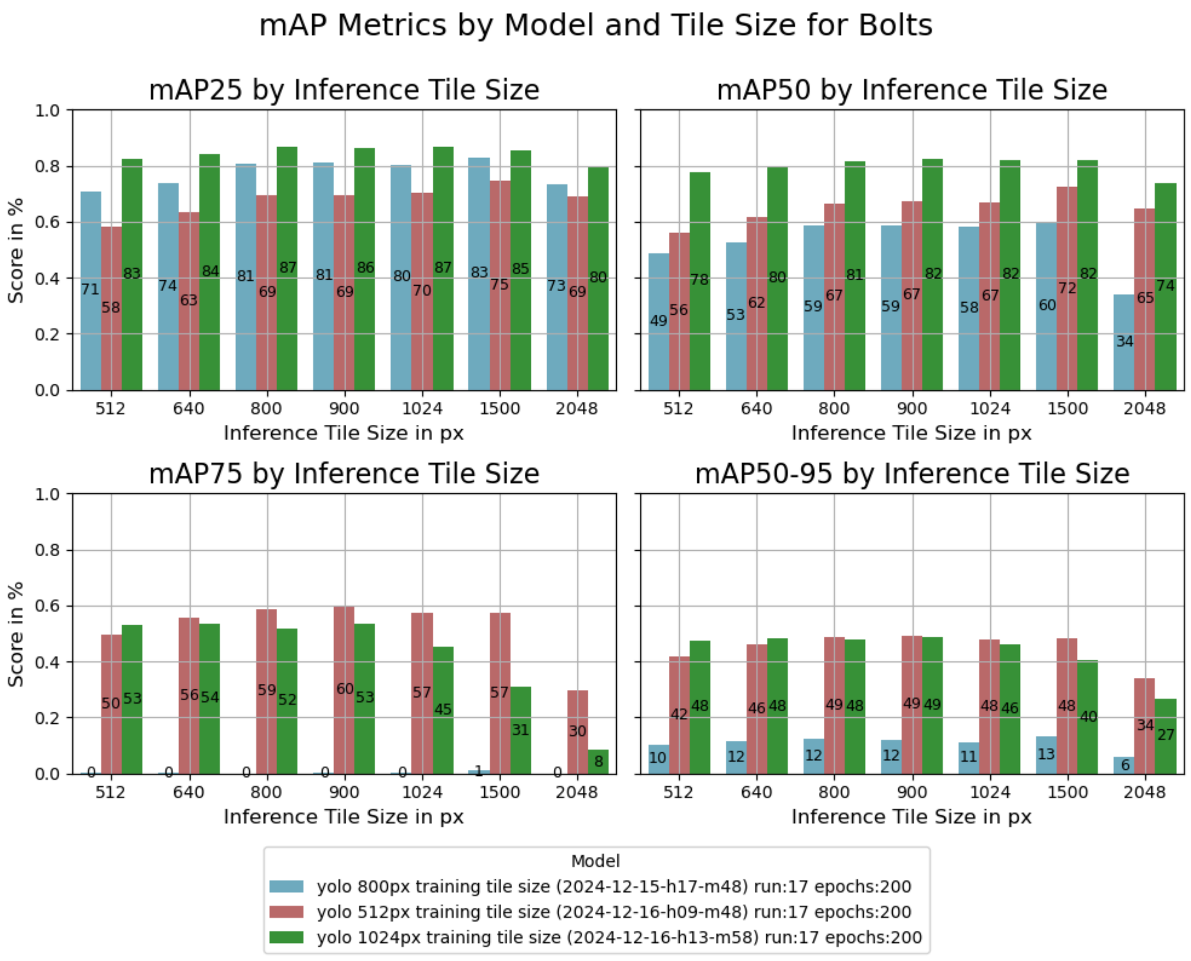


Figure 143: Tile Size Comparison for Bolts

III 3.2.1.0.0.1 F1 Score vs Confidence Threshold

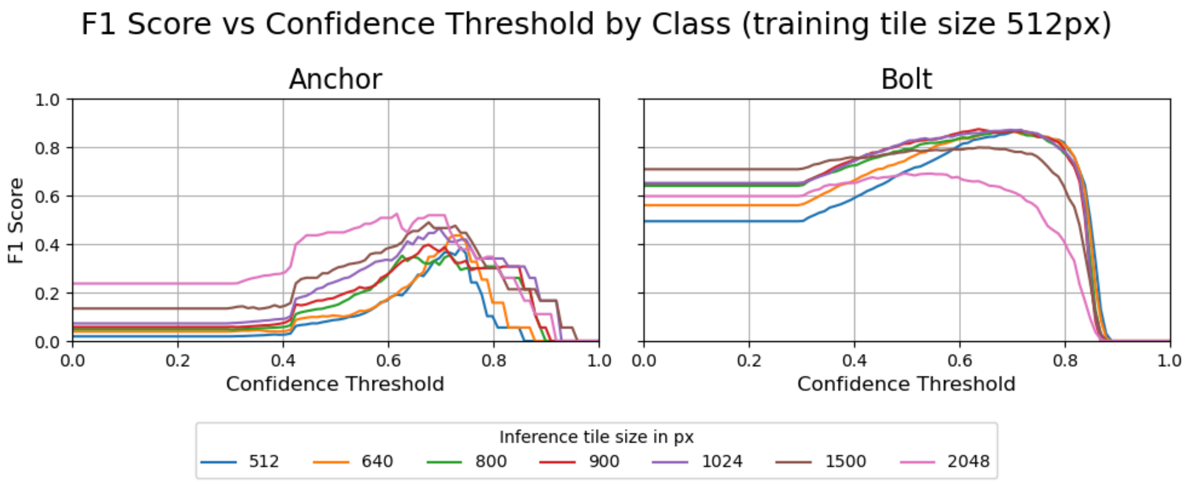


Figure 144: F1 Score vs Confidence Threshold for 512px Tile Size

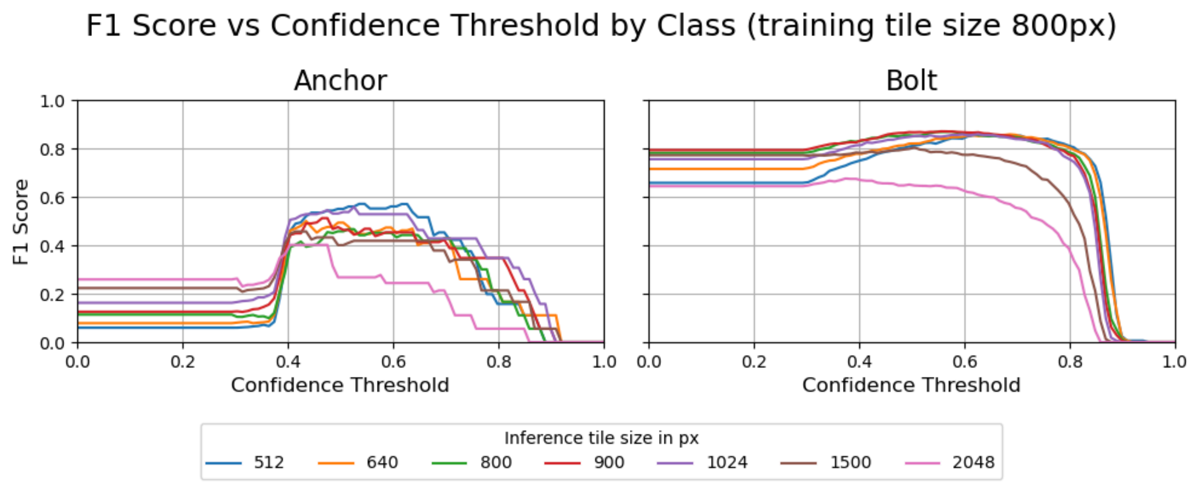


Figure 145: F1 Score vs Confidence Threshold for 800px Tile Size

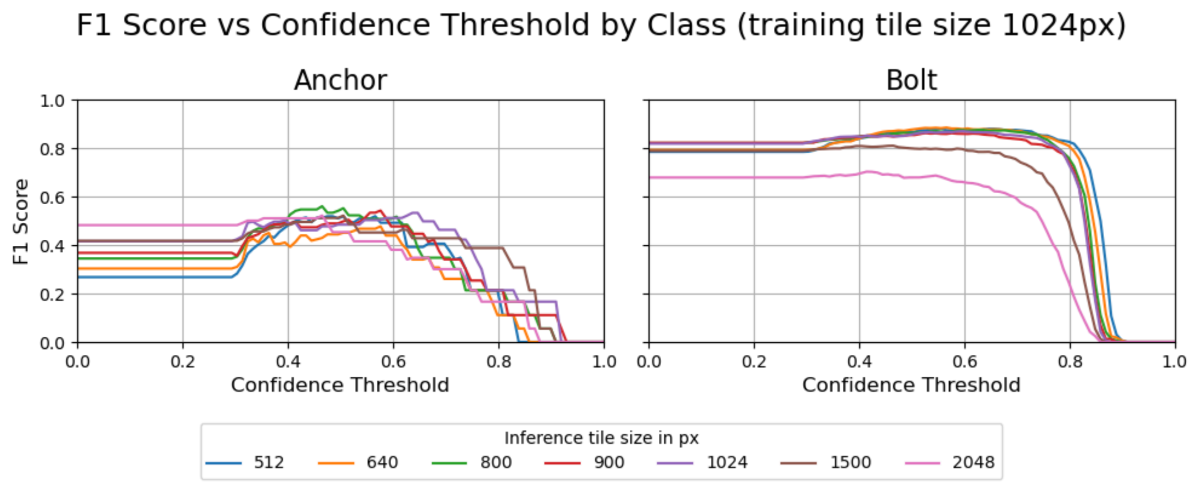


Figure 146: F1 Score vs Confidence Threshold for 1024px Tile Size

III 3.2.1.0.0.2 Precision vs Recall

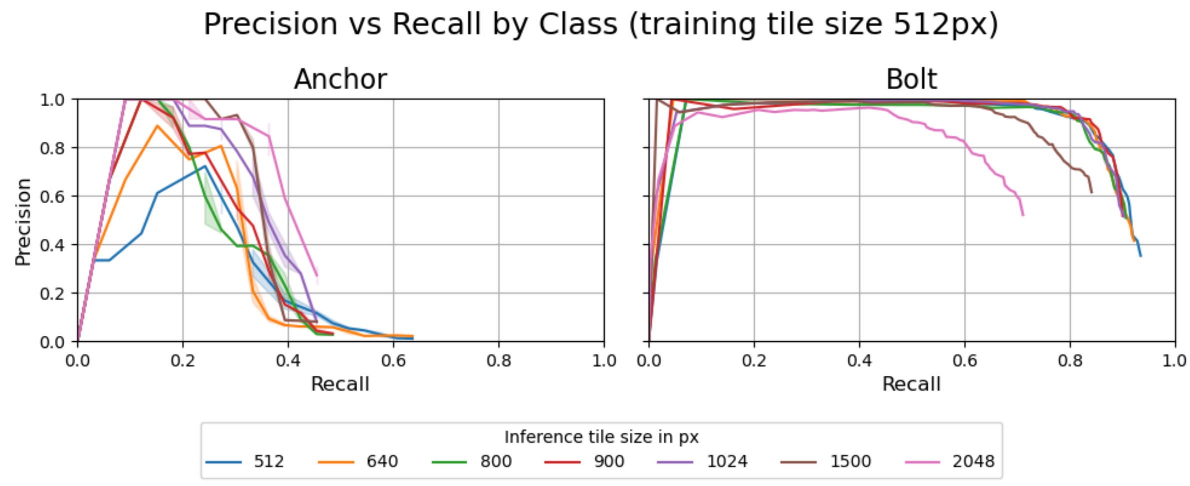


Figure 147: Precision vs Recall for 512px Tile Size

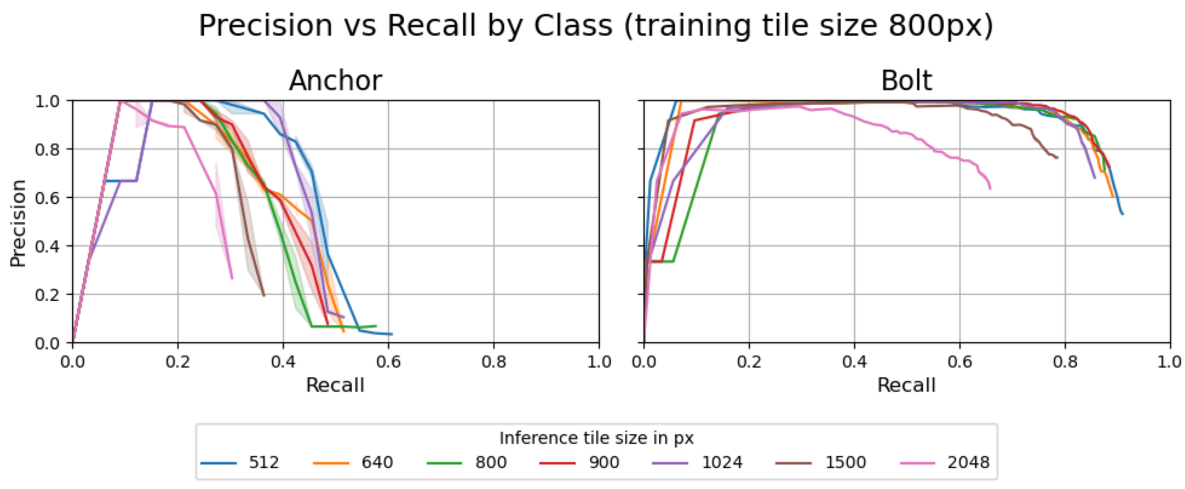


Figure 148: Precision vs Recall for 800px Tile Size

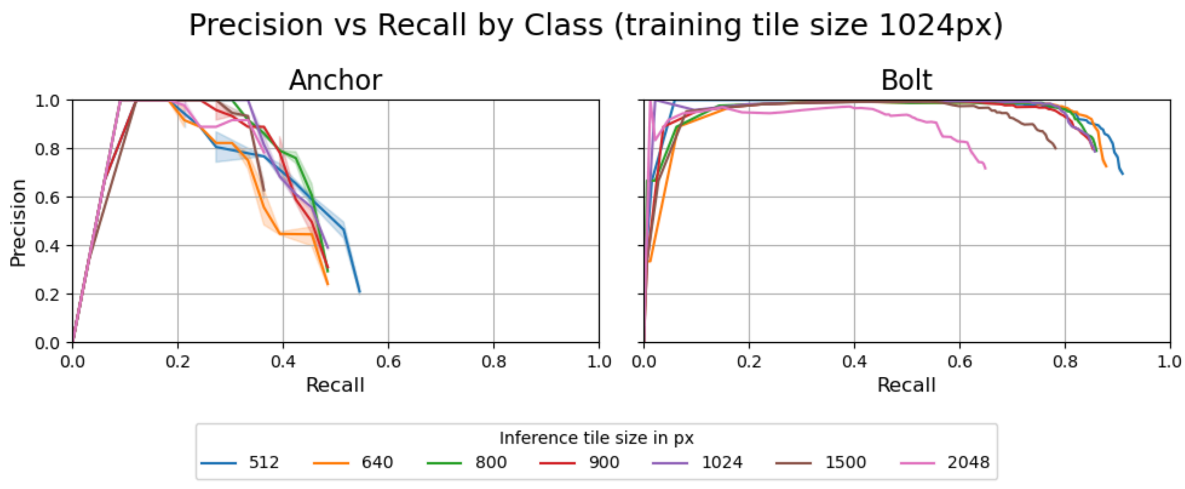


Figure 149: Precision vs Recall for 1024px Tile Size

III 3.2.2 Precision vs Confidence Threshold

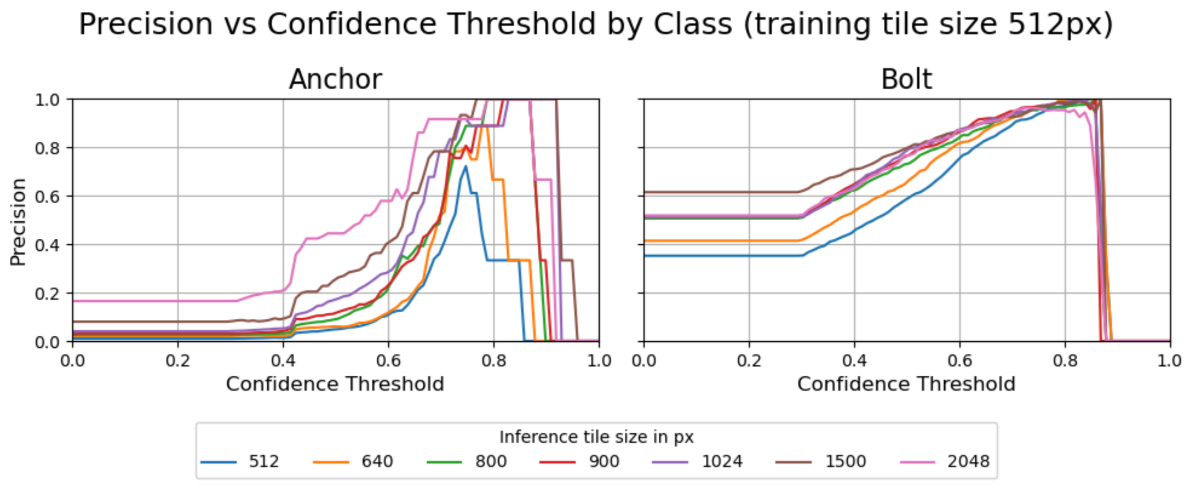


Figure 150: Precision vs Confidence Threshold for 512px Tile Size

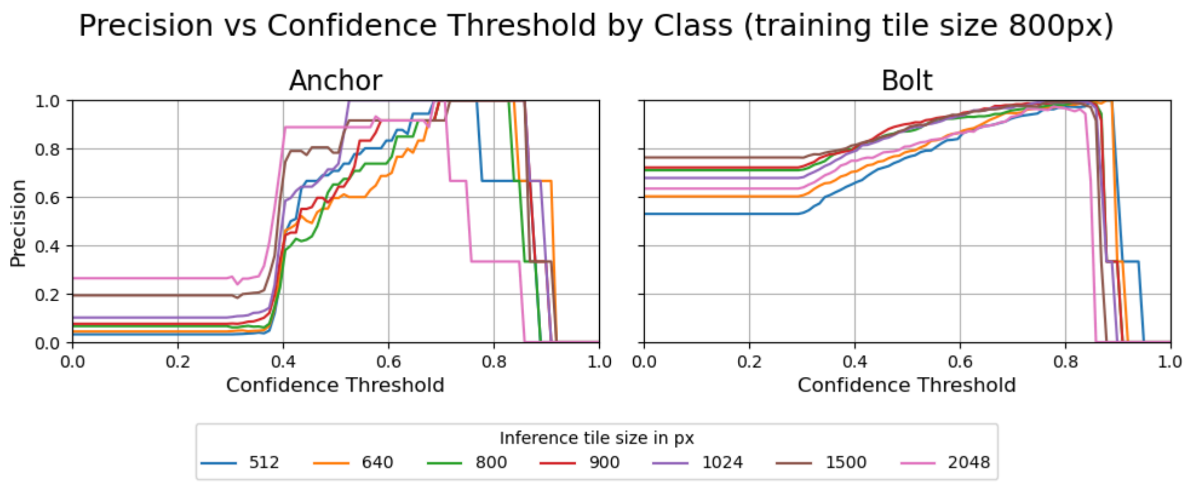


Figure 151: Precision vs Confidence Threshold for 800px Tile Size

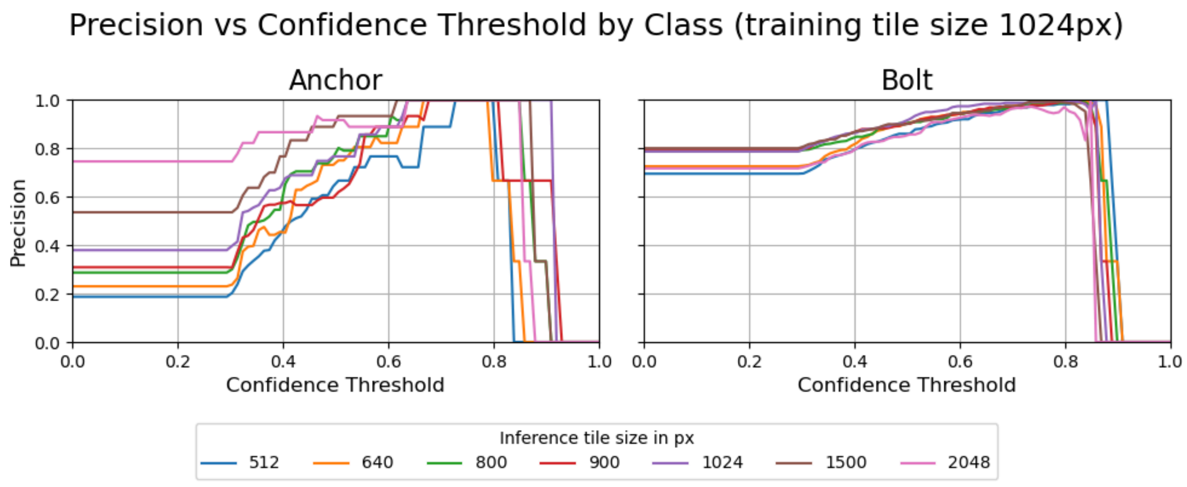


Figure 152: Precision vs Confidence Threshold for 1024px Tile Size

III 3.2.3 Recall vs Confidence Threshold

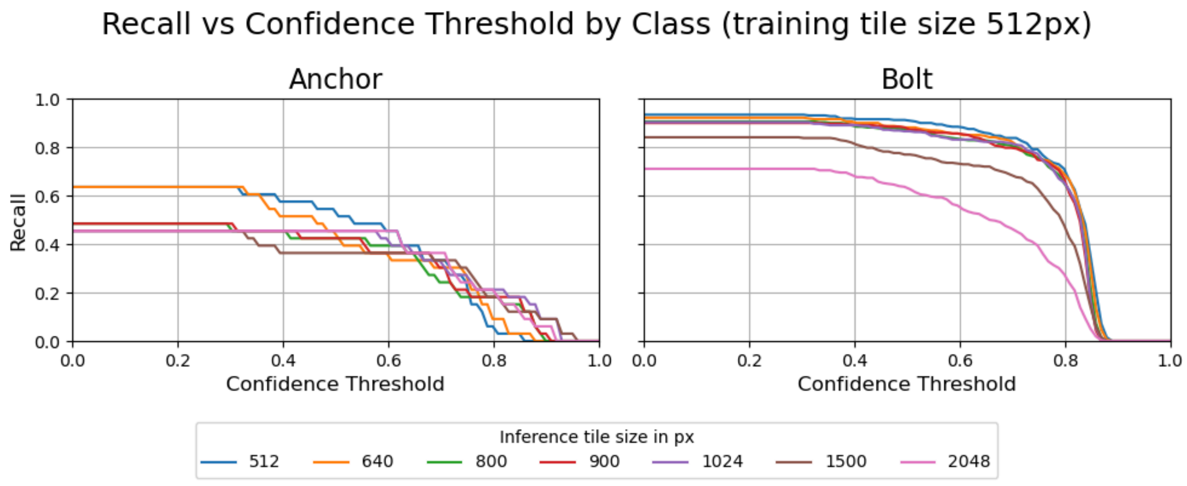


Figure 153: Recall vs Confidence Threshold for 512px Tile Size

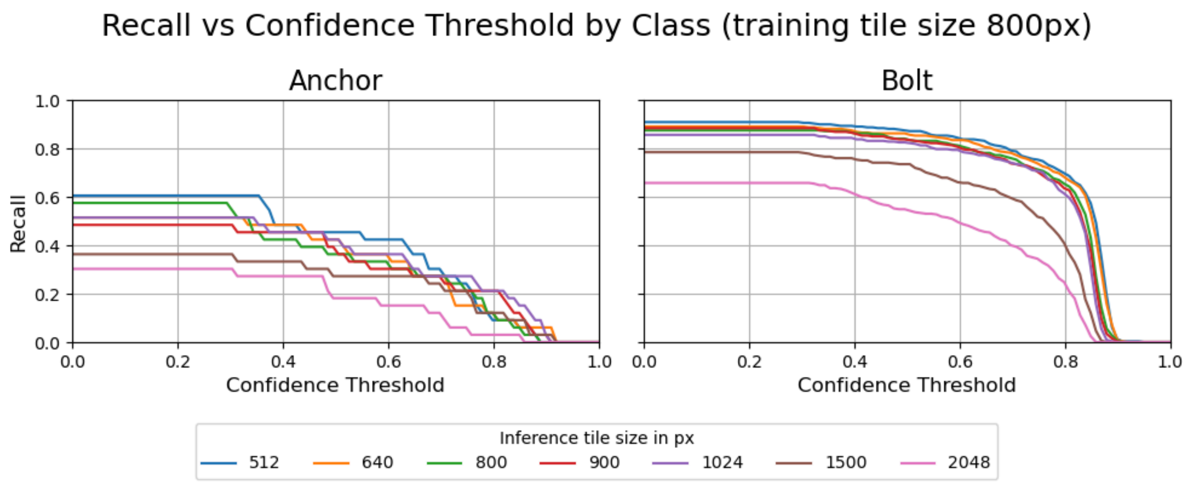


Figure 154: Recall vs Confidence Threshold for 800px Tile Size

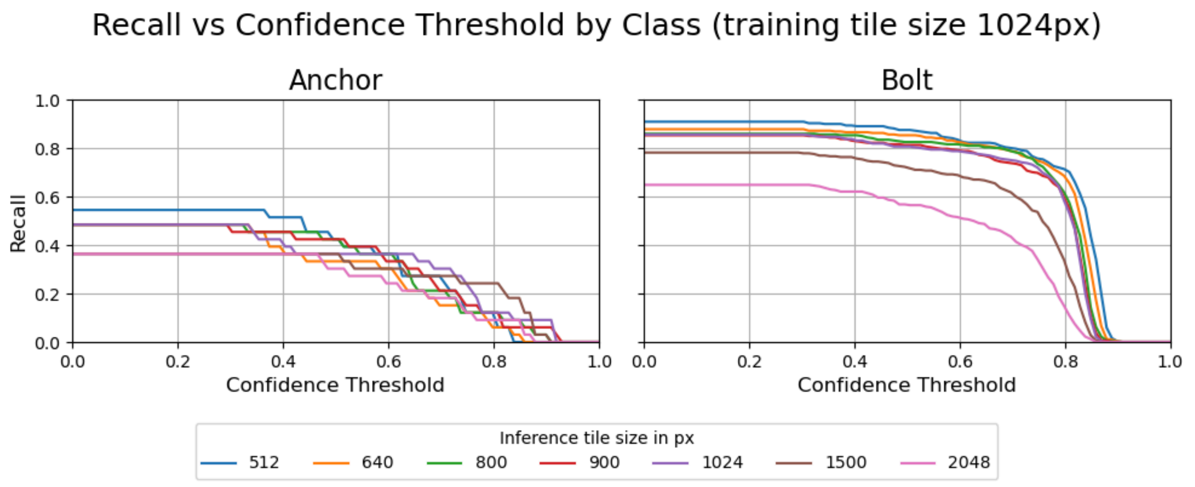


Figure 155: Recall vs Confidence Threshold for 1024px Tile Size

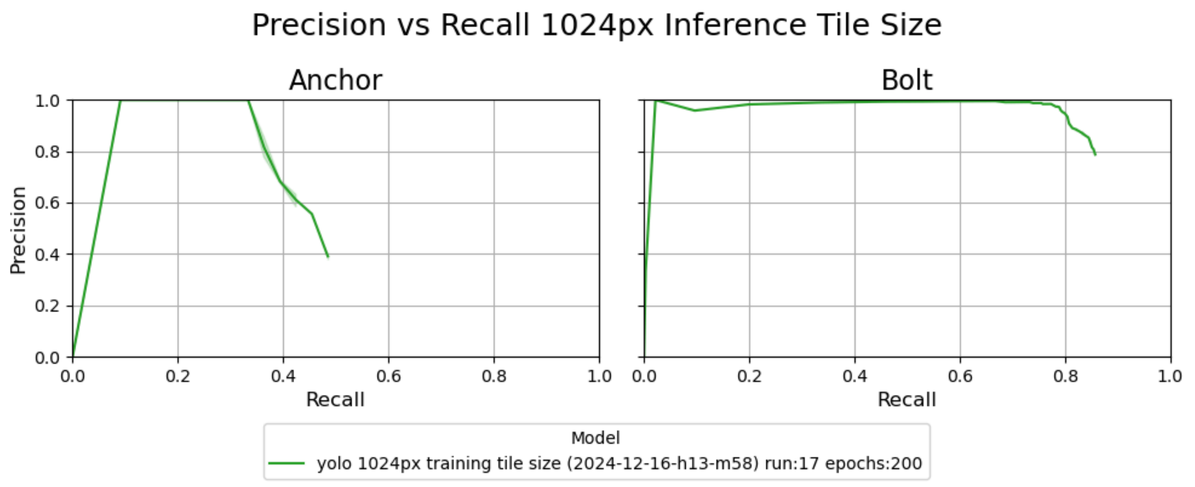


Figure 156: Precision vs Recall (1024px Inference Tile Size)

III 4 Screenshots and Various Snippets

This sections contains screenshots of the implemented software and used tools.

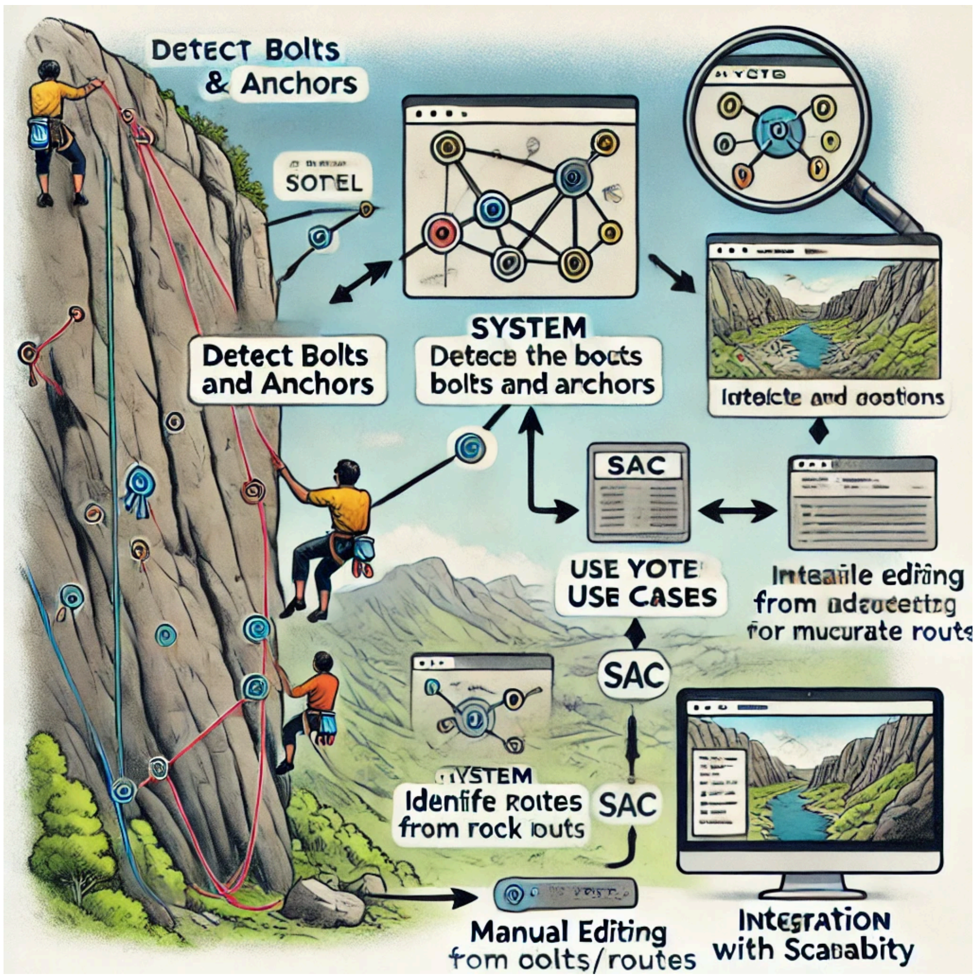
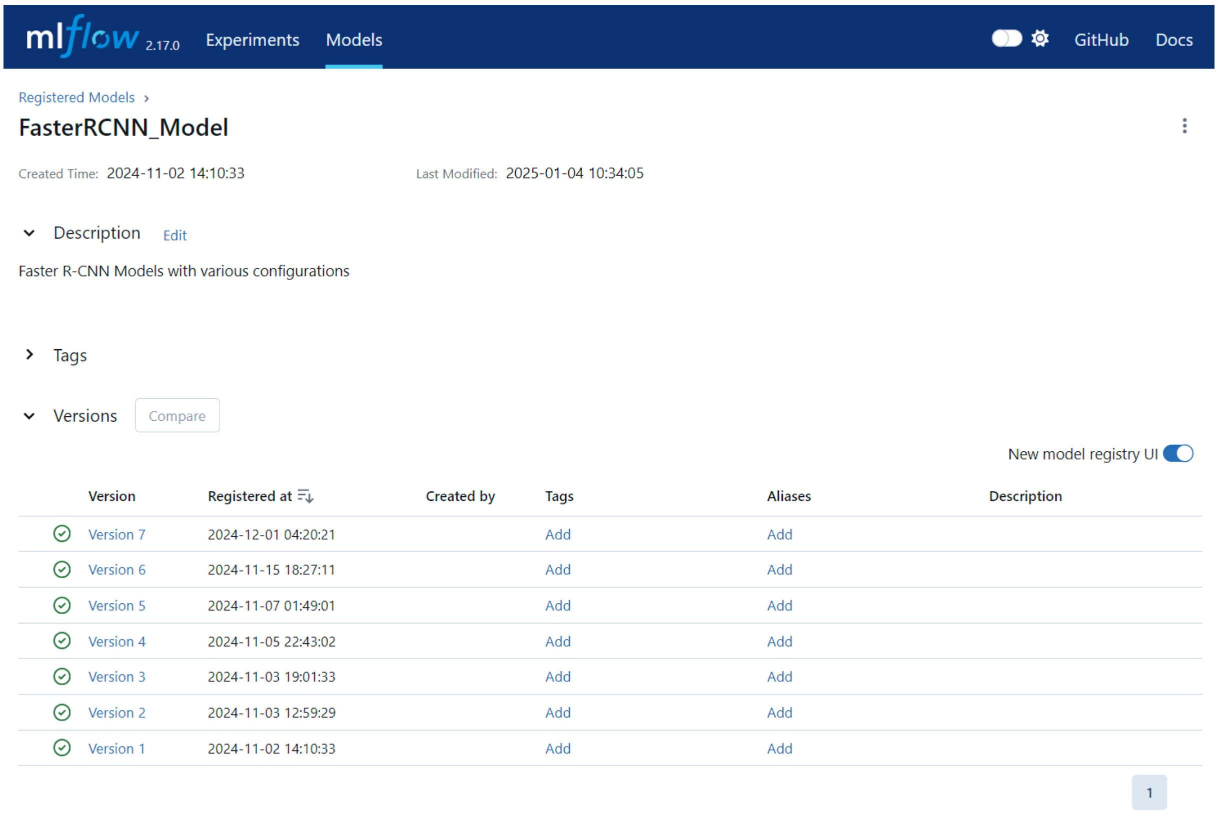

Detect Bolts & Anchors
SOTEL
Detect Bolts and Anchors
SYSTEM Detece the bocts bolts and archors
Intelcte and ootions
SAC
USE YOTE USE CASES
Inteanle edring from adeoeettrg for mucurate routs
SAC
SYSTEM Idenfife Roites from rock iouts
SAC
Manual Editing fom oolts/routes
Integration with Scaoabity

Figure 157: BoltFinder interpretation by ChatGPT

III 4.1 MLFlow

MLFlow R-CNN model registry



Registered Models ›

FasterRCNN_Model

Created Time: 2024-11-02 14:10:33   Last Modified: 2025-01-04 10:34:05

Description Edit

Faster R-CNN Models with various configurations

Tags

Versions Compare

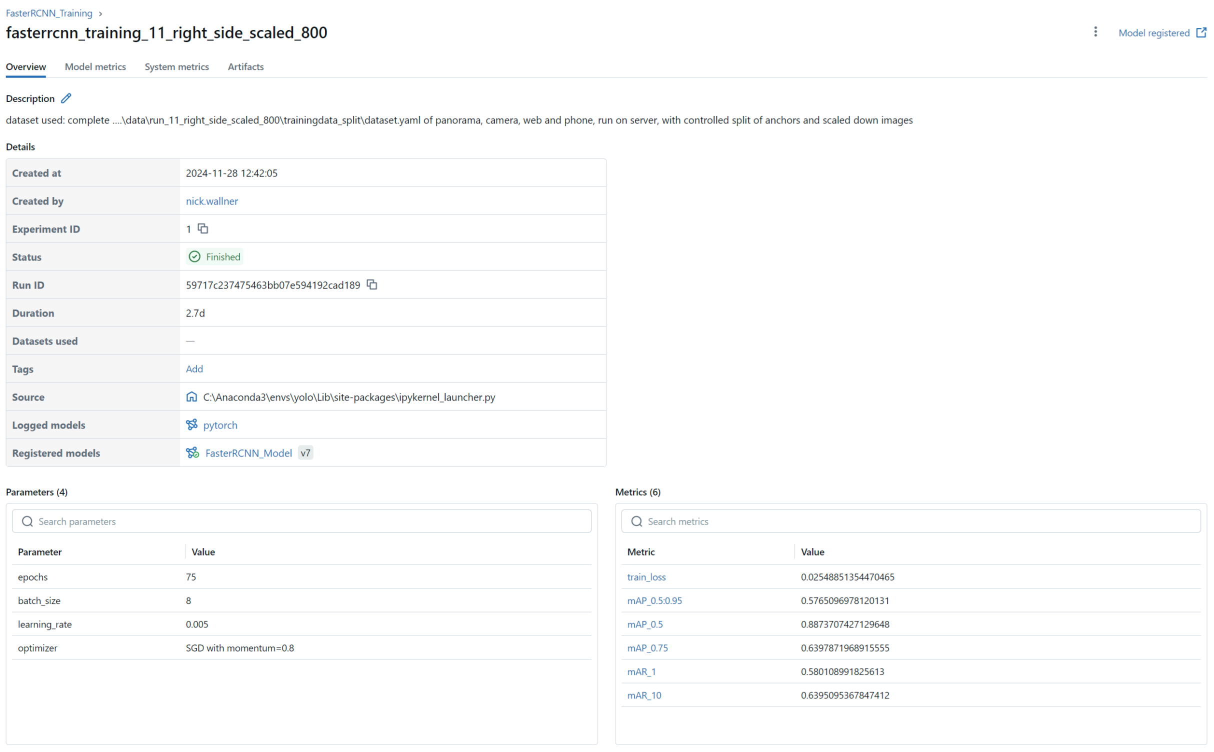
New model registry UI

	Version	Registered at	Created by	Tags	Aliases	Description
✓	Version 7	2024-12-01 04:20:21		Add	Add	
✓	Version 6	2024-11-15 18:27:11		Add	Add	
✓	Version 5	2024-11-07 01:49:01		Add	Add	
✓	Version 4	2024-11-05 22:43:02		Add	Add	
✓	Version 3	2024-11-03 19:01:33		Add	Add	
✓	Version 2	2024-11-03 12:59:29		Add	Add	
✓	Version 1	2024-11-02 14:10:33		Add	Add	

1

Figure 158: MLFlow Faster R-CNN model registry

R-CNN model with MLFlow tracking. The model is trained with the Faster R-CNN model and the MLFlow tracking tool.



FasterRCNN_Training ›

fasterrcnn_training_11_right_side_scaled_800

Description

dataset used: complete ....\data\run_11_right_side_scaled_800\trainingdata_split\dataset.yaml of panorama, camera, web and phone, run on server, with controlled split of anchors and scaled down images

Details

Created at	2024-11-28 12:42:05
Created by	nick.wallner
Experiment ID	1
Status	Finished
Run ID	59717c237475463bb07e594192cad189
Duration	2.7d
Datasets used	—
Tags	Add
Source	C:\Anaconda3\envs\yolo\Lib\site-packages\ipykernel_launcher.py
Logged models	pytorch
Registered models	FasterRCNN_Model v7

Parameters (4)

Search parameters

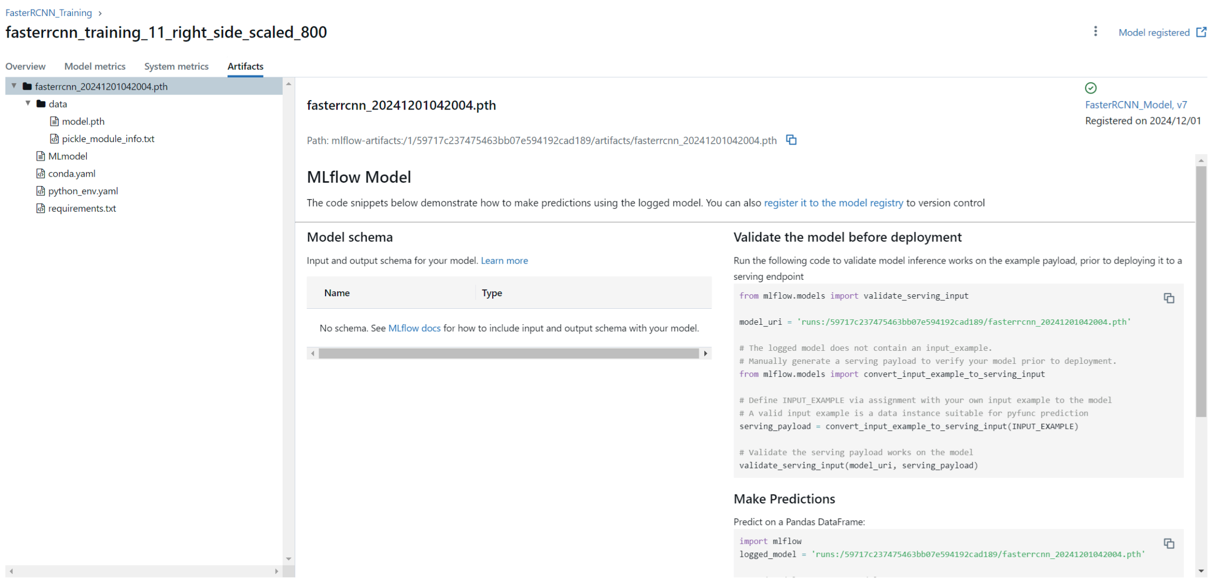
Parameter	Value
epochs	75
batch_size	8
learning_rate	0.005
optimizer	SGD with momentum=0.8

Metrics (6)

Search metrics

Metric	Value
train_loss	0.02548851354470465
mAP_0.5:0.95	0.5765096978120131
mAP_0.5	0.8873707427129648
mAP_0.75	0.6397871968915555
mAR_1	0.580108991825613
mAR_10	0.6395095367847412

Figure 159: MLFlow Faster R-CNN model



FasterRCNN_Training ›

fasterrcnn_training_11_right_side_scaled_800

Model registered

Overview Model metrics System metrics Artifacts

- fasterrcnn_20241201042004.pth
  - data
    - model.pth
    - pickle_module_info.txt
  - MLmodel
  - conda.yaml
  - python_env.yaml
  - requirements.txt

fasterrcnn_20241201042004.pth

FasterRCNN_Model, v7
Registered on 2024/12/01

Path: mlflow-artifacts:/1/59717c237475463bb07e594192cad189/artifacts/fasterrcnn_20241201042004.pth

MLflow Model

The code snippets below demonstrate how to make predictions using the logged model. You can also register it to the model registry to version control

Model schema

Input and output schema for your model. Learn more

Name	Type
No schema. See MLflow docs for how to include input and output schema with your model.	

Validate the model before deployment

Run the following code to validate model inference works on the example payload, prior to deploying it to a serving endpoint

```
from mlflow.models import validate_serving_input

model_uri = 'runs:/59717c237475463bb07e594192cad189/fasterrcnn_20241201042004.pth'

# The logged model does not contain an input_example.
# Manually generate a serving payload to verify your model prior to deployment.
from mlflow.models import convert_input_example_to_serving_input

# Define INPUT_EXAMPLE via assignment with your own input example to the model
# A valid input example is a data instance suitable for pyfunc prediction
serving_payload = convert_input_example_to_serving_input(INPUT_EXAMPLE)

# Validate the serving payload works on the model
validate_serving_input(model_uri, serving_payload)
```

Make Predictions

Predict on a Pandas DataFrame:

```
import mlflow
logged_model = 'runs:/59717c237475463bb07e594192cad189/fasterrcnn_20241201042004.pth'
```

Figure 160: MLFlow Faster R-CNN model folder

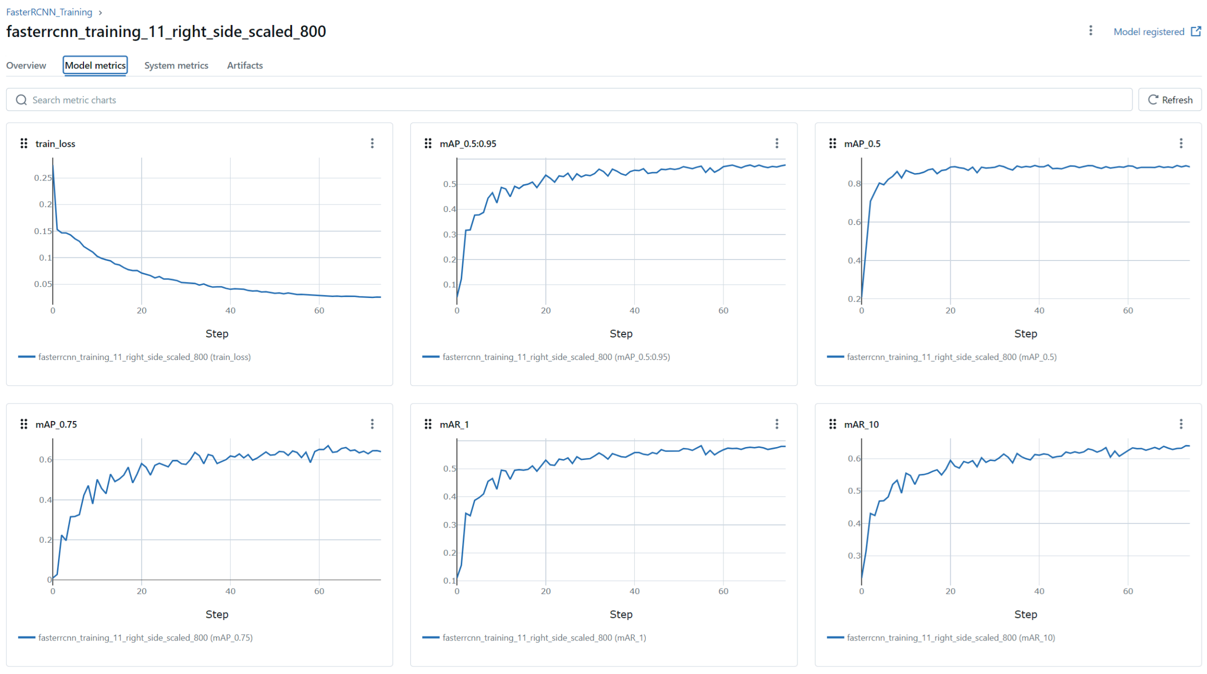


Figure 161: MLFlow Faster R-CNN model metrics

III 4.2 Algorithm Voronoi

Another possible approach is to split the points into a **Voronoi** and split the Voronoi into clusters. The clusters are then used to find the routes. This would simplify the algorithm and make it more robust. The split is done by always taking the path which ends up lower and starting at the highest bolt. Since this sometimes also segregates one route into two, the approach is not ideal Figure 163.

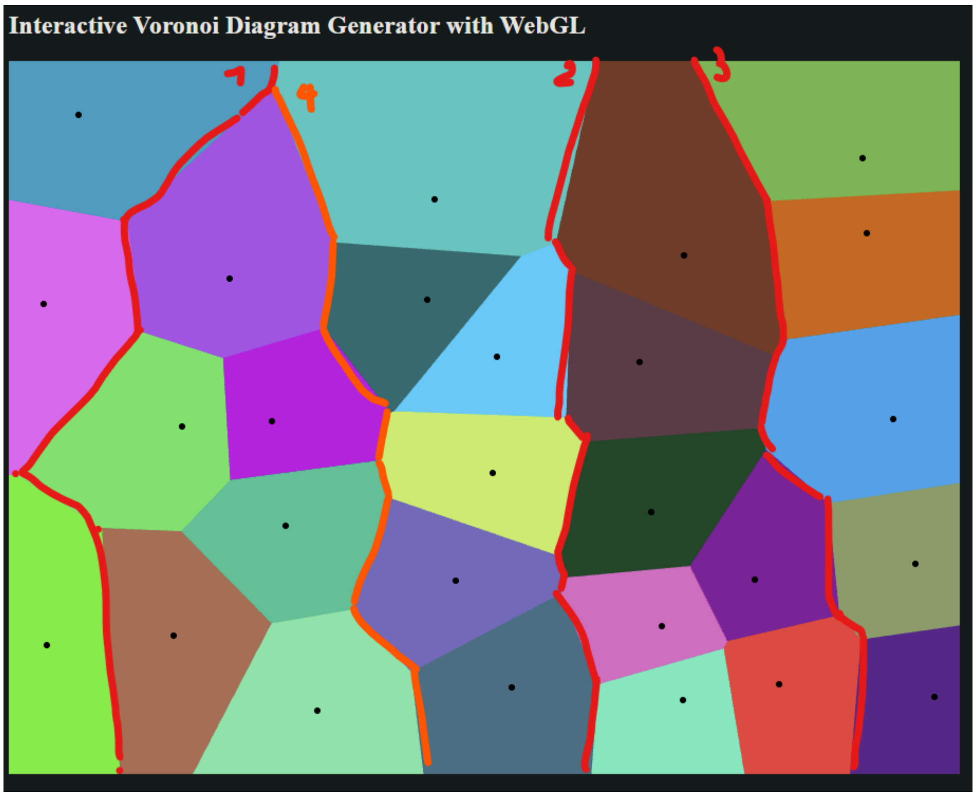


Figure 162: Voronoi path segmentation

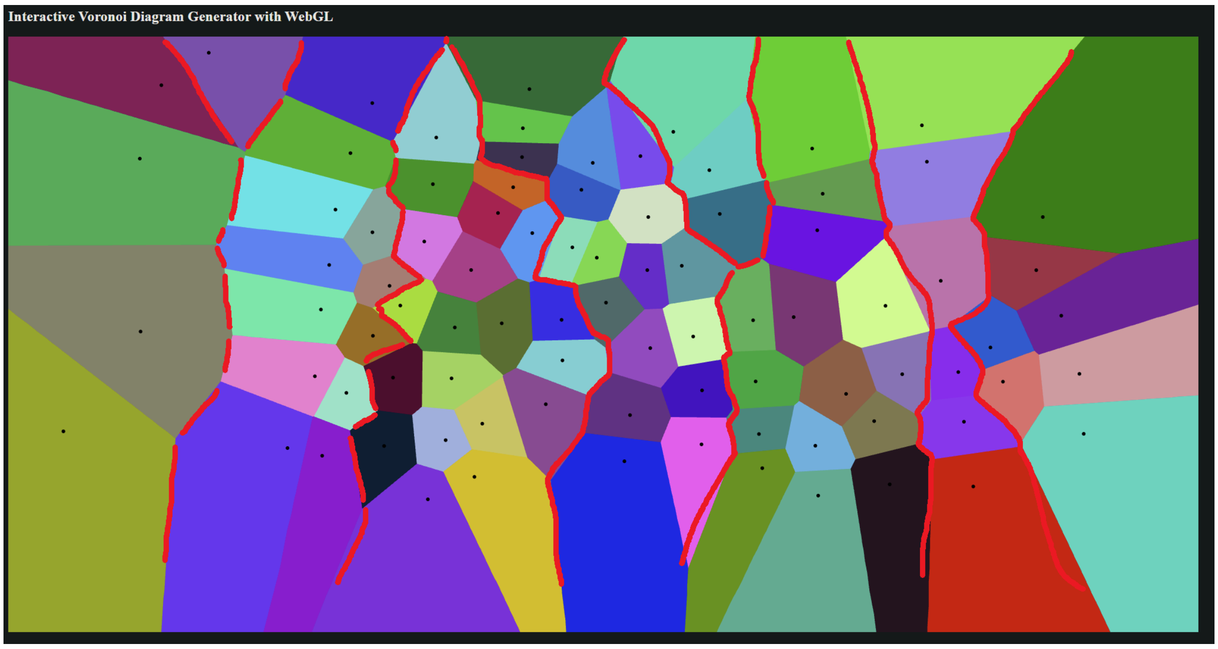


Figure 163: Problems with the Voronoi path segmentation

III 4.3 Dataset

The following diagrams show the size distribution of the annotations in the dataset.

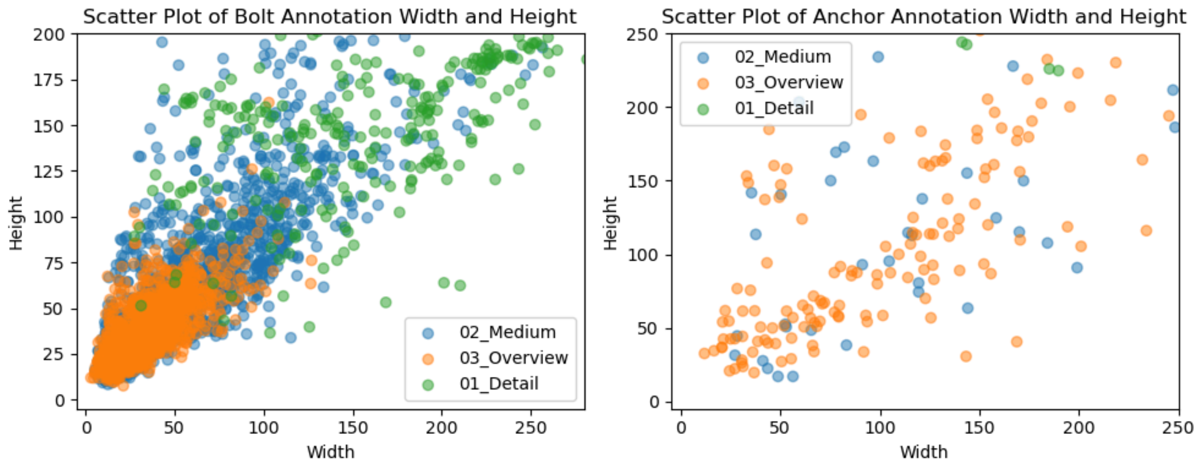


Figure 164: Annotation sizes in the dataset without outliers

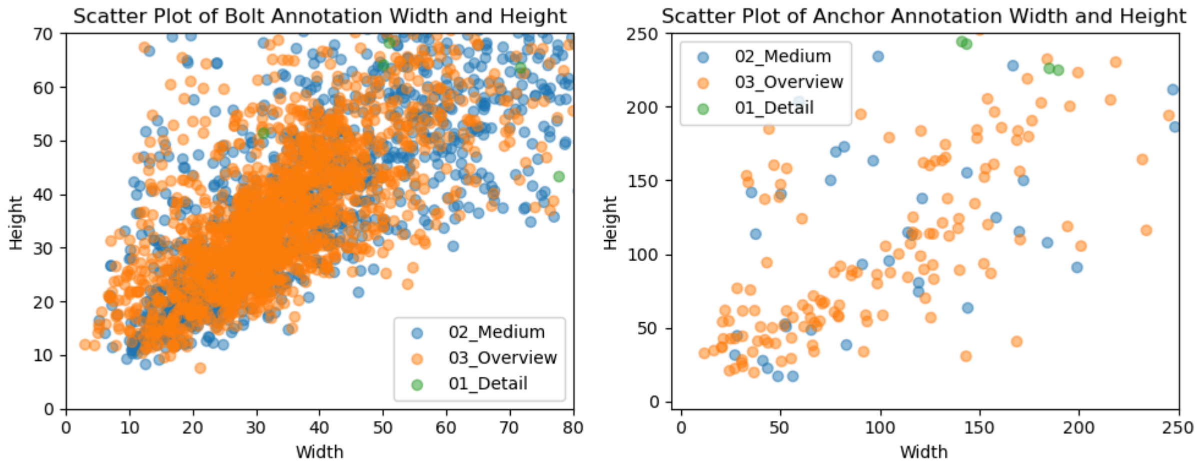
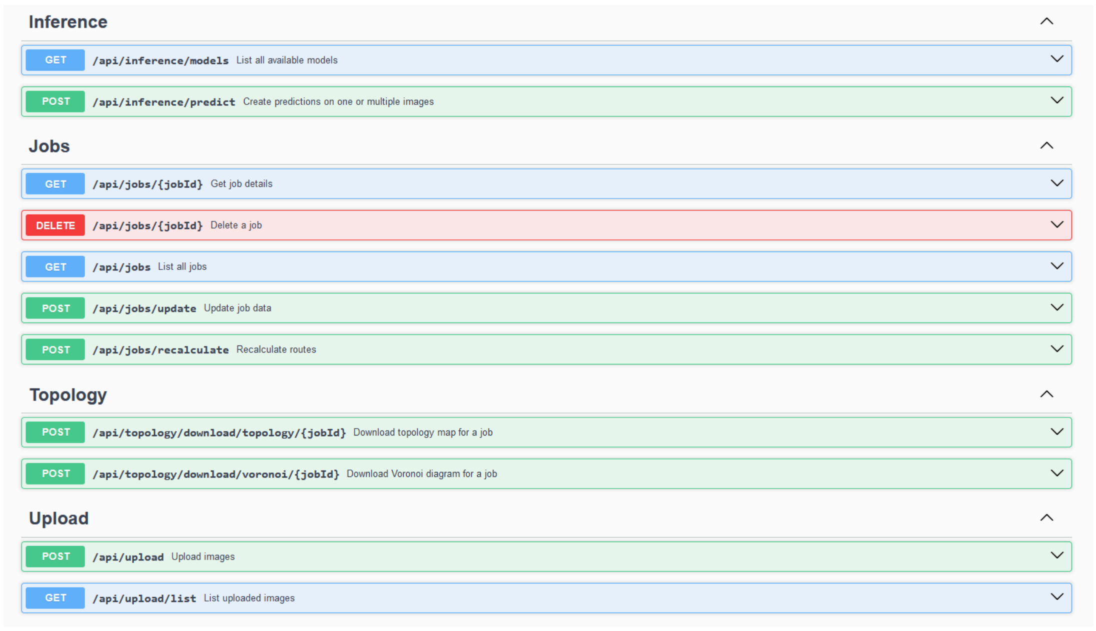


Figure 165: Zoomed Annotation sizes in the dataset

III 5 Swagger Documentation

Both the backend API and the Inference API provide comprehensive Swagger documentation for their respective endpoints. These documents serve as a reference for understanding and interacting with the available APIs.

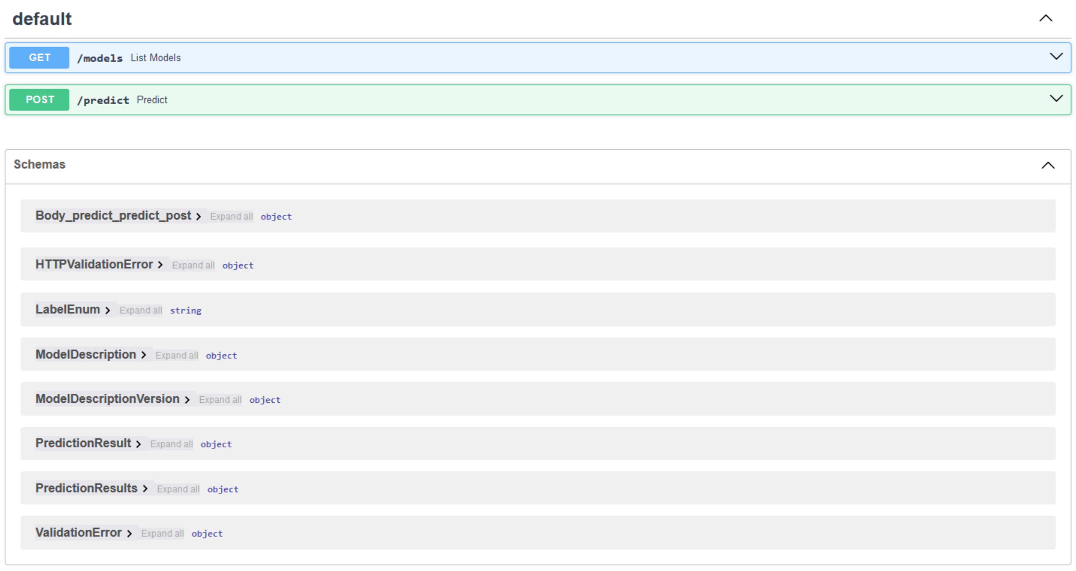

Inference
GET `/api/inference/models` List all available models
POST `/api/inference/predict` Create predictions on one or multiple images

Jobs
GET `/api/jobs/{jobId}` Get job details
DELETE `/api/jobs/{jobId}` Delete a job
GET `/api/jobs` List all jobs
POST `/api/jobs/update` Update job data
POST `/api/jobs/recalculate` Recalculate routes

Topology
POST `/api/topology/download/topology/{jobId}` Download topology map for a job
POST `/api/topology/download/voronoi/{jobId}` Download Voronoi diagram for a job

Upload
POST `/api/upload` Upload images
GET `/api/upload/list` List uploaded images

Figure 166: Swagger documentation for the Backend API


default
GET `/models` List Models
POST `/predict` Predict

Schemas
Body_predict_predict_post > Expand all object
HTTPValidationError > Expand all object
LabelEnum > Expand all string
ModelDescription > Expand all object
ModelDescriptionVersion > Expand all object
PredictionResult > Expand all object
PredictionResults > Expand all object
ValidationError > Expand all object

Figure 167: Swagger documentation for the Inference API

III 6 Database ERD

The database ERD shows the structure of the database and the relationships between the tables. The ERD is shown in Figure 168.

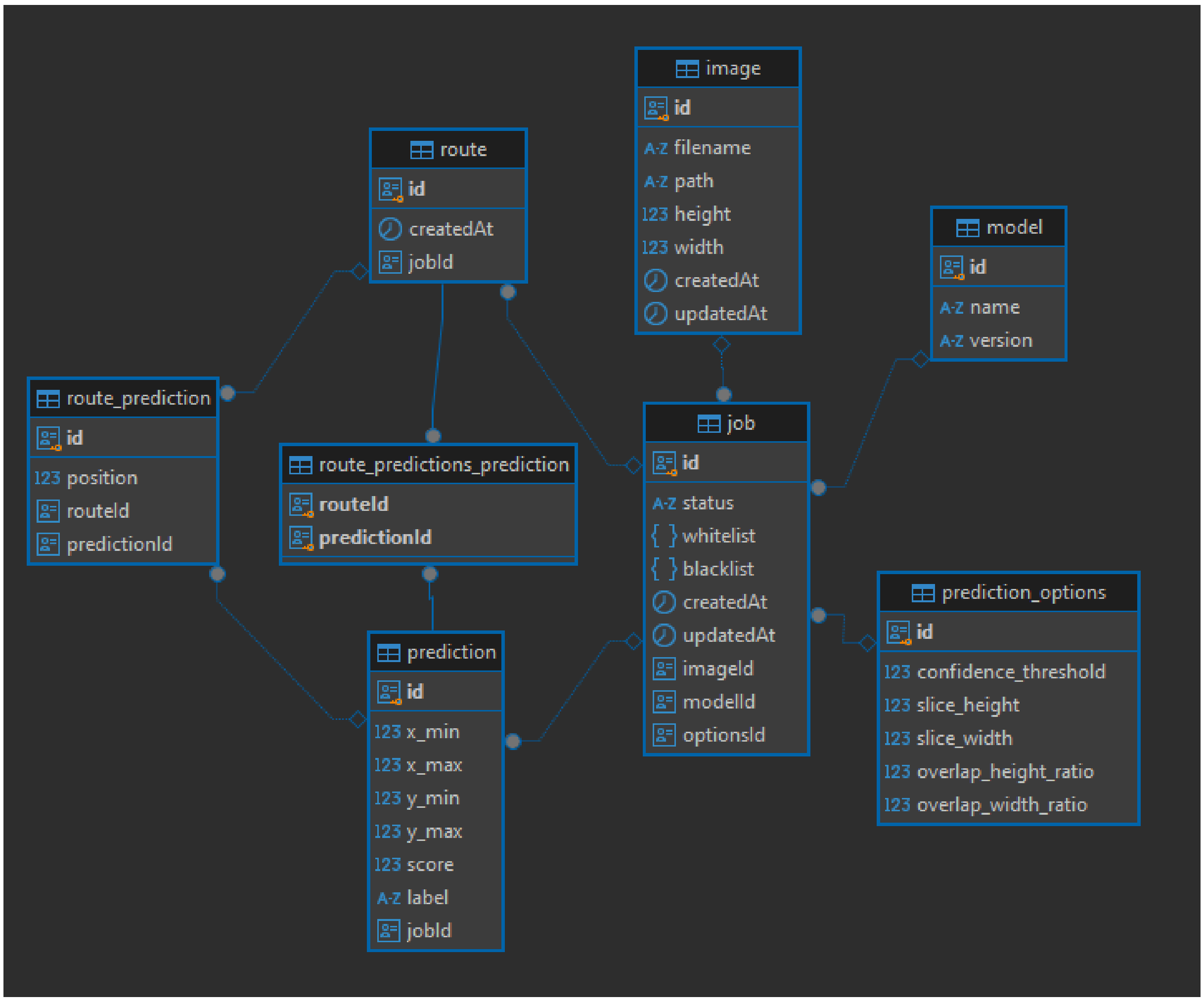


Figure 168: Database ERD

IV Appendix B

1 Glossary

Glossary A list of terms and definitions used in the project.

1.1 Technical Terms

- **Object Detection**: A computer vision technique that involves identifying and locating objects within an image or video. It is commonly used in applications such as surveillance, autonomous vehicles, and image retrieval.
- **Machine Learning (ML)**: A subset of artificial intelligence that enables systems to learn and improve from experience without being explicitly programmed. It involves algorithms that can learn from and make predictions or decisions based on data.
- **SAHI**: Slicing and Handling Images (SAHI) is a library that divides large images into smaller overlapping tiles for object detection. It helps address challenges related to detecting small objects and handling edge cases in image processing.
- **YOLO**: You Only Look Once (YOLO) is a real-time object detection system that processes images in a single pass, making it faster than traditional two-stage methods. It divides the image into a grid and predicts bounding boxes and class probabilities for each grid cell.
- **Faster R-CNN**: Faster Region-based Convolutional Neural Network (Faster R-CNN) is a two-stage object detection model that uses a region proposal network to generate candidate object regions and a detection network to classify and refine the proposals.
- **RPN**: Region Proposal Network (RPN) is a neural network that generates region proposals for object detection. It predicts bounding boxes and object scores for potential objects in an image.
- **DETR**: Detection Transformer (DETR) is a transformer-based object detection model that directly predicts object bounding boxes and class labels without using anchor boxes or region proposal networks.
- **DINO**: Data-Driven Instance Noise for Unsupervised Representation Learning (DINO) is a self-supervised learning method that uses instance discrimination to learn visual representations without manual annotations.
- **IoU**: Intersection over Union (IoU) is a metric used to evaluate the accuracy of object detection algorithms. It measures the overlap between the predicted bounding box and the ground truth bounding box.
- **Bounding Box (BBox)**: A rectangular box that surrounds an object in an image. It is used to define the location and size of the object within the image. Usually contains locations of the four box corners.
- **POC**: Proof of Concept (POC) is a demonstration to validate the feasibility of a concept or idea. It is used to test the viability of a project before investing resources in full-scale development.
- **Computational Geometry**: A branch of computer science that deals with the study of algorithms and data structures for geometric problems. It includes topics such as convex hulls, Voronoi diagrams, and Delaunay triangulations.

- **Voronoi Diagram**: A partitioning of a plane into regions based on the distance to a specific set of points. Each region contains all points closer to a specific point than to any other point in the set.
- **Clustering**: A machine learning technique that involves grouping similar data points together based on their features or attributes. It is commonly used for data analysis, pattern recognition, and image segmentation.
- **Convex Hull**: The smallest convex polygon that encloses a set of points in a plane. It is used in computational geometry for various applications, such as collision detection and spatial indexing.
- **UI**: User Interface (UI) refers to the visual elements and controls that users interact with in software applications. It includes components like buttons, menus, and input fields that enable users to interact with the system.
- **SAC**: The Swiss Alpine Club (SAC) is the largest mountaineering and climbing organization in Switzerland. It promotes alpine sports, mountain protection, and outdoor activities in the Swiss Alps.
- **API**: Application Programming Interface (API) is a set of rules and protocols that allow different software applications to communicate with each other. It defines the methods and data formats used for interaction between systems.
- **IMUs**: Inertial Measurement Units (IMUs) are electronic devices that measure and report a body's specific force, angular rate, and sometimes the magnetic field surrounding the body. They are commonly used in motion tracking, navigation, and robotics.

1.2 Climbing Terms

- **Climbing route cataloging / digitization**: this describes the process of documenting and digitizing the routes created in outdoor climbing sectors into a clear, organized format.
- **Topo**: Short for topographic map, a climbing topo is a visual representation of a climbing route or sector, typically showing the path and location of climbing routes on a rock wall. Topos can have various levels of detail, from simple line drawings to detailed photographs with route descriptions.
- **Climbing Sector**: A designated area within a climbing gym or outdoor climbing area that contains a set of routes or problems.
- **Top**: the "top" can refer to the highest point of a route (top-out) or the method of climbing known as top roping, where the rope is anchored at the top of the route, allowing the climber to ascend with safety while being belayed from below.
- **Anchor (Stand)**: A fixed point of protection that a climber attaches to for safety, typically consisting of two or more securely bolted points in the wall for redundancy. Anchors can also be natural features like trees or serve as a belay stations.
- **Bolts**: Fixed hardware drilled into the rock, typically used for sport climbing. Bolts can include hanger bolts (which have a loop for attaching a quickdraw) and expansion bolts (which expand within the rock to secure themselves).
- **Glue-in Bolts**: These are a type of fixed protection that are installed using a special epoxy resin. They are commonly used in sport climbing and for bolting new routes. The bolt itself is placed into a hole drilled into the rock, and then a strong adhesive is used to secure it in place.
- **Slings**: Loops of webbing used for anchoring, extending gear placements, or creating a harness.
- **Quickdraws**: Connectors that allow climbers to attach the rope to bolts or other gear. They consist of two carabiners (one fixed and one movable) and a sewn sling.
- **Carabiner**: a metal loop with a spring-loaded gate used to quickly and reversibly connect components in safety-critical systems.
- **Hooks**: Often referred to in the context of aid climbing. They are typically used to hook onto features in the rock and provide a point of protection. There are various types of hooks, such as daisy hooks, pitons, and cam hooks.
- **Cams**: Devices that expand within a crack to secure a climber.
- **Nuts**: Wedge-shaped pieces of metal that can be placed in cracks to provide protection.
- **Hexes**: Similar to nuts but with a larger, hexagonal shape for placement in wider cracks.
- **Belay device**: a mechanical device used to control a rope during belaying (the act of managing the rope to protect a climber from falling).
- **Beta**: information about a climbing route, including the sequence of moves, holds, and techniques required to complete.

2 Table of Figures

3 Table of Tables

4 Table of Code Fragments

5 Bibliography

[1] "SAC Year Report 2023." Accessed: Nov. 29, 2024. [Online]. Available: https://www.sac-cas.ch/fileadmin/Der_SAC/%C3%9Cber_uns/PDF/2023_Jahresbericht_SAC_final.pdf

[2] "SAC Sektionen." Accessed: Nov. 29, 2024. [Online]. Available: https://www.sac-cas.ch/de/der-sac/sektionen/

[3] "Modulbeschreibung: Bachelorarbeit Informatik." Accessed: Nov. 25, 2024. [Online]. Available: https://studien.ost.ch/allModules/41626_M_BAI21.html

[4] "Computer Vision Based Indoor Rock Climbing Analysis," Sep. 27, 2024, *ucsd*. Accessed: Dec. 16, 2023. [Online]. Available: https://kastner.ucsd.edu/ryan/wp-content/uploads/sites/5/2022/06/admin/rock-climbing-coach.pdf

[5] "ClimbSense - Automatic Climbing Route Recognition using Wrist-worn Inertia Measurement Units," Sep. 27, 2024, *CHI 2015, Crossings, Seoul, Korea*. Accessed: Dec. 16, 2023. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/2702123.2702311

[6] S. Ekaireb, M. A. KHAN, P. PATHURI, P. H. BHATIA, R. SHARMA, and N. MANJUNATH-MURKAL, "Computer vision based indoor rock climbing analysis." Accessed: Sep. 30, 2024. [Online]. Available: https://kastner.ucsd.edu/ryan/wp-content/uploads/sites/5/2022/06/admin/rock-climbing-coach.pdf

[7] "Deep Learning-Based Object Detection Techniques for Remote Sensing Images: A Survey," Sep. 27, 2024, *mdpi*. Accessed: Dec. 16, 2023. [Online]. Available: https://www.mdpi.com/2072-4292/14/10/2385

[8] R. Andersson Dickfors and N. Grannas, "Object Detection Using Deep Learning on Metal Chips in Manufacturing." Accessed: Jun. 22, 2021. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-55068

[9] "Metal Surface Defect Detection Using Modified YOLO," Dec. 16, 2023, *mdpi*. Accessed: Sep. 27, 2024. [Online]. Available: https://www.mdpi.com/1999-4893/14/9/257

[10] "Metal surface defect detection based on improved YOLOv5," Nov. 27, 2023, *Scientific Reports*. Accessed: Sep. 30, 2024. [Online]. Available: https://doi.org/10.1038/s41598-023-47716-2

[11] "Object Detection and Classification of Metal Polishing Shaft Surface Defects Based on Convolutional Neural Network Deep Learning," Jan. 02, 2020, *mdpi*. Accessed: Sep. 30, 2024. [Online]. Available: https://www.mdpi.com/2076-3417/10/1/87

[12] "Deep learning of rock images for intelligent lithology identification," Jul. 01, 2021, *Computers & Geosciences*. Accessed: Sep. 30, 2024. [Online]. Available: https://doi.org/10.1016/j.cageo.2021.104799

[13] viet, *Metal Dataset*. Roboflow. Accessed: Dec. 06, 2024. [Online]. Available: https://universe.roboflow.com/viet-km12g/metal-yv9u3

[14] James Gallagher, "How to Detect Metal Defects with Computer Vision." Accessed: Dec. 25, 2024. [Online]. Available: https://blog.roboflow.com/detect-metal-defects/

[15] J. Terven, D.-M. Córdova-Esparza, and J.-A. Romero-González, “A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS,” *MDPI AG*. Accessed: Jan. 01, 2025. [Online]. Available: http://dx.doi.org/10.3390/make5040083

[16] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.” Accessed: Nov. 06, 2024. [Online]. Available: https://arxiv.org/abs/1506.01497

[17] L. Huang *et al.*, “Multi-Scale Feature Fusion Convolutional Neural Network for Indoor Small Target Detection.” Accessed: Nov. 30, 2024. [Online]. Available: https://www.frontiersin.org/journals/neurorobotics/articles/10.3389/fnbot.2022.881021

[18] A. Mertan, D. J. Duff, and G. Unal, “Single image depth estimation: An overview,” *Elsevier BV*. Accessed: Nov. 26, 2024. [Online]. Available: http://dx.doi.org/10.1016/j.dsp.2022.103441

[19] Santiago Garrido, Mohamed Abderrahim, and Luis Moreno, “PATH PLANNING AND NAVIGATION USING VORONOI DIAGRAM AND FAST MARCHING.” Accessed: Dec. 10, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S147466701638538X

[20] “Key words for use in RFCs to Indicate Requirement Levels,” Mar. 01, 1997, *Harvard University*. Accessed: Sep. 27, 2024. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119

[21] “UML And Patterns,” May 01, 1997, *craiglarman*. Accessed: Sep. 27, 2024. [Online]. Available: https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf

[22] “ISO/IEC 25010,” May 01, 1997. Accessed: Sep. 27, 2024. [Online]. Available: https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[23] “Focal Loss for Dense Object Detection,” 2017, *IEEE International Conference on Computer Vision (ICCV)*. Accessed: Dec. 16, 2023. [Online]. Available: https://arxiv.org/abs/1708.02002

[24] “End-to-End Object Detection with Transformers,” 2020, *ECCV*. Accessed: Dec. 16, 2023. [Online]. Available: https://ai.meta.com/research/publications/end-to-end-object-detection-with-transformers/

[25] “Microsoft Image Composite Editor.” Accessed: Dec. 30, 2024. [Online]. Available: https://www.microsoft.com/en-us/research/project/image-composite-editor/

[26] “Hugin.” Accessed: Dec. 30, 2024. [Online]. Available: https://hugin.sourceforge.io/

[27] “Ultralytics YOLOv11.” Accessed: Jan. 04, 2024. [Online]. Available: https://docs.ultralytics.com/models/yolo11/

[28] Ahmed Fawzy Gad and James Skelton, “Faster R-CNN Explained for Object Detection Tasks.” Accessed: Oct. 04, 2024. [Online]. Available: https://www.digitalocean.com/community/tutorials/faster-r-cnn-explained-object-detection

[29] F. C. Akyon, S. O. Altinuc, and A. Temizel, “Slicing Aided Hyper Inference and Fine-Tuning for Small Object Detection.” Accessed: Oct. 11, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9897990

[30] Alex Beutel, “Voronoi Diagram WebGL.” Accessed: Dec. 30, 2024. [Online]. Available: https://alexbeutel.com/webgl/voronoi.html

[31] E. Stefanakis, “Web Mercator and Raster Tile Maps: Two Cornerstones of Online Map Service Providers,” Jun. 2017.

[32] N. Guo, W. Xiong, Q. WU, and N. Jing, "An Efficient Tile-Pyramids Building Method for Fast Visualization of Massive Geospatial Raster Datasets," Jan. 01, 2016. Accessed: Jan. 05, 2025. [Online]. Available: https://www.researchgate.net/figure/Sketch-map-of-tile-pyramid-structure_fig3_311423420

[33] C. Zhang, L. Yang, M. Liu, and J. Wu, "Improved texture rendering based on the MIPMAP algorithm."