June 2024

Bachelors Thesis

AI-Pipeline Integration for Historic Plan Digitisation

Leveraging transformer based deep learning models in a custom desktop app to automate the digitisation of landscaping plans

Kevin Löffler
kevin.loeffler@ost.ch

Advisor:
Prof. Dr. Mitra Purandare
mitra.purandare@ost.ch
OST

Co-Examiner:
Dr. Marc Röösli
marc.roeoesli@mt.com
Mettler-Toledo International Inc.

Eastern Switzerland University of Applied Sciences

Abstract

This thesis proposes a full-stack solution for the digitisation process of the Swiss Archive for Landscape Architecture (ASLA).

A desktop app is developed using Tauri, enabling the management of the archive's data as well as the correction of AI predictions.

The AI pipeline is containerised using Docker and is accessible via a web API. Each plan is formatted and preprocessed, before three deep learning models are applied: a pretrained layout model (LayoutLMv3) to detect all text occurrences with k-means clustering to group text boxes into logical blocks, and a transformer-based OCR model (TrOCR) to extract text. Relevant entities are then identified using a custom-trained German BERT model. The output undergoes post-processing for formatting and normalisation, with project-specific keywords like the architect's name filtered out. The predicted metadata is sent back to the client app where metadata files track all changes to the image, ensuring non-destructive editing.

Every weekend, the machine learning models are retrained on all the manually changed predictions. The thesis focuses more on implementing a robust pipeline and continuous retraining than on improving the models because continuous retraining is expected to enhance the AI pipeline's performance over time.

The app significantly speeds up the digitisation process for the archive and is a substantial improvement over the old Excel-based workflow. The AI pipeline's prediction accuracy varies by model. Marker detection is 100% reliable, and the OCR model reaches 98% accuracy after retraining on only 77 images. The NER model, currently at 46% accuracy, is about 10% better than the model from the SA project. If the accuracy continues to increase with additional training data, an F1-score of over 80% can be foreseen with 600 images. Thanks to the new app, these images can be collected in less than a month of archival work.

Keywords: Applied Machine Learning, Transformer Based Deep Learning, Software Developement, Desktop App, AI Integration, Continuous Retraining

Management Summary

The Archive of Swiss Landscape Architecture (ASLA) holds over 100′000 documents, primarily plans from prominent Swiss and European architects of the 20th century. These documents are used for teaching, exhibitions, and architectural projects. Currently, the archive manually digitises its documents, a process that is highly time-consuming.

This thesis aims to automate the digitisation process. The solution involves two main components: a transformer-based deep learning and image processing pipeline to locate and extract text, identify relevant entities such as the client, date, or scale of a plan, and a desktop app to manually edit and refine the AI-generated output.

The AI pipeline employs different machine learning models based on the transformer architecture, as well as classical image processing techniques. It predicts entities that the archive currently transcribes by hand. The pipeline is continuously retrained with new data as it becomes available, increasing its accuracy over time.

The app handles project management and uploads the images to the AI pipeline. It receives the predictions and displays them to the user, who can edit and correct them where necessary. Besides managing the entities, the app also offers basic image editing functionality like changing the white balance, contrast, and brightness of an image.

The new system significantly improves the efficiency of the digitisation process. It provides a robust and user-friendly interface for archivists, reducing the manual effort required and ensuring consistent metadata collection. With more training data, the AI pipeline can be refined on the archive's data, ultimately saving the archive years in their effort to digitise their plans and make them accessible to the world faster.

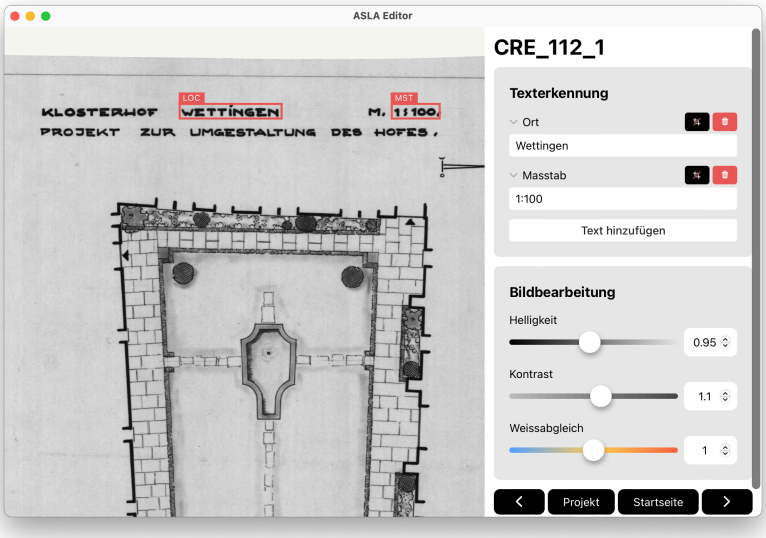

Figure 1: The new editor with predictions from the AI pipeline

Contents

1 Introduction

The Archive of Swiss Landscape Architecture (ASLA) [1] collects and preserves plans from prominent Swiss and European architects of the 20th century. Currently, the archive holds over 100′000 documents, organised into estates gifted by the architects themselves or their families. These documents provide insights into architectural practices and landscape design of the last century.

These plans serve multiple purposes: they are used in teaching at OST, displayed in exhibitions, and accessed by external architects and planners working on various projects. However, a significant challenge is that most of these plans have not been digitised, and some estates lack even a basic projects list. Consequently, whenever someone wants to view a specific plan, a member of the archive staff must manually search through all the plans from a particular architect to locate relevant materials, which is both time-consuming and inefficient.

To address these issues and enhance the accessibility and usability of the archive, a decision was made to digitise the collection and publish them on the archives website. A dedicated photostation was constructed where plans can be laid on a large board, secured with magnets, and photographed by a permanently installed camera. The images are automatically synced with a computer, where metadata is collected. This metadata includes crucial information such as the project client (Bauherr), the location of the building, the scale of the plan, and the creation date. Additionally, the images are cropped to the size of the plan and straightened if necessary. Despite these advancements, the process remains highly manual and involves frequent switching between various tools, including the camera app, image viewer, file explorer, Photoshop, and Excel. The metadata collection in Excel, in particular, is cumbersome and not user-friendly, making the entire digitisation process very time consuming and expensive.

During my SA project, I investigated various automation and machine learning techniques to determine if aspects of this process could be automated. We concluded that a deep learning pipeline could significantly speed up the metadata collection. However, due to insufficient training data, the models could not be effectively retrained, resulting in unsatisfactory accuracy.

In this bachelor's thesis, I aimed to integrate this AI pipeline into a production environment. Specifically, I developed a desktop application designed to streamline the process of working with the plans, managing projects, and collecting metadata. This application is supported by an enhanced version of the aforementioned AI pipeline, which is continuously retrained as new data becomes available. This iterative improvement aims to increase the accuracy and efficiency of metadata collection, ultimately making the archive more accessible and functional for all users.

2 Objective

2.1 Vision

The ultimate goal of this thesis is to provide the archive with a new and improved workflow for digitisation. To address the user experience challenges associated with metadata collection, a frontend application should be developed. This application is designed to facilitate the organisation of images into projects (estates) and provide a visual editor where the collected metadata can be displayed and edited directly on the plan. The visual editor will also allow for image adjustments that were previously done in Photoshop, such as white balance, contrast, and brightness adjustments. Furthermore, the app will have access to a custom-trained AI pipeline (based on the findings from the SA) that automates both the image editing and metadata collection processes. This new workflow aims to be faster, more consistent, and more enjoyable than the current one. By integrating advanced machine learning techniques and a user-friendly interface, the digitisation process will become significantly more efficient, reducing the manual effort required and enhancing the overall user experience.

2.2 Assignment

The following tasks where defined with the thesis supervisor Mitra Purandare at the start of the project:

1. Build an app that can be used by the archive employee:
   - Define functional and non-functional requirements based on the needs of the archive
   - Write user stories based on the needs of the archive
   - Build a desktop GUI app that runs on the archives computer
   - Import the images from the camera and send them to the server
   - Receive the processed images from the server and display them with the prediction results
   - The user can edit or override the predicted metadata
   - *The user can delete, edit or draw new bounding boxes for possible future model training*
   - *Add basic image editing capabilities (whitebalance, exposure, contrast) to manualy tweak the settings*
2. Deploy the AI pipeline on a server
   - Containerise the AI pipeline and make it accessible via REST API (send image, recive prediction)
   - Add preprocessing to the AI pipeline:
     - Marker detection and cropping
     - Image enhancements with whitebalance, exposure, contrast
     - *Dynamic* image processing to make the image better suited for the AI pipeline like denoising, thresholding, distance transforms and opening morphological operations
   - Build a test suite that measures the pipeline performance to make changes to it measureable and compareable

- Try to improve the performance of the specific models and pipeline as a whole
- Compare inference performance on different server hardware (CPU / GPU)
- *Implement continous retraining and model improvement with new data*
- *Visualise model evolution*
- *Image stiching for plans that do not fit in a single photograph*

Italic: Optional task

While all required and most optional tasks could be completed, visualisation of model improvement and image stitching where not possible in the given time.

2.3 App Specification

The editor is a crossplatform desktop app that enables a user to interact with the AI-pipeline on the server. It is responsible for managing the images captured by the camera, send them to the server and save the response accordingly. The following workflows have been defined with help from Simon Orga, the technical supervisor of the archive:

Capture

1. Open App
2. Start the upload (runs continuously in the background)
3. Start taking pictures, rename them manually and move them to upload directory
   - Handle images without project -> warning: user should create new project
4. Receive response and save the images automatically to the working directory
   - Handle possible pipeline errors

Edit

1. Open project
2. Select image to edit
3. Change predicted metadata if necessary
4. Change bounding boxes if necessary
5. Edit image: whitebalance, contrast, exposure
6. Export and upload all images to the server (supervisor)

2.4 Requirements

According to the assignment and the app specification, the following requirements where defined:

ID	Scope	Task	Description
FR-1	App	Projects	The app should load the projects, keywords, upload directory, api endpoint and other configuration data from a global enviorenment file on launch. If no enviorenment file is specified or found the user should be asked to provide one or guided through the project of creating a new one.

ID	Scope	Task	Description
FR-2	App	Projects	The app should display a welcome screen with a list of projects.
FR-3	App	Projects	Users should be able to create and delete projects, specifying details such as the project code, architect's name, working directory, and export directory. Changes should be synced to the global enviorenment.
FR-4	App	Upload	The app should provide an upload screen to start/stop uploads and handle errors. The upload should run continuously in the background.
FR-5	App	Upload	The app should not allow the upload of images without a project.
FR-6	App	Upload	The app should automatically receive responses from the server and save images to the working directory.
FR-7	App	Upload	Atomicity and data integrity should be guaranteed when moving image files and the corresponding metadata.
FR-8	App	Editor	The app should have a visual editor where the user can draw bounding boxes, edit the labels and text of entities, and adjust the image.
FR-9	App	Editor	All edits to an image should be performed non-destructively. All changes should be stored as instructions in a json file and saved with the image.
FR-10	App	Editor	All changes should be autosaved. The metadata file and image in the working directory act as a single source of truth.
FR-11	App	Editor	All manual changes should be recorded for later retraining of the models.
FR-12	App	Editor	The editor should keep track of the last edited image so that when a project is reopened, the user is at the same point where they left off.
FR-13	App	User	The functionality of the app should be well documented so that new users can be onboarded quickly and with minimal human guidance.
FR-14	AI	Server	The api endpoints should always be reachable. A crash or error in the pipeline should not lead to api failure.
FR-15	AI	Server	If no requests are made for 30 minutes, the ai-models should be unloaded from memory to free up unused resources. The are reloaded if a new request is made.
FR-16	AI	PreProc.	The user placed markers should be recognised and the image cropped accordingly.

ID	Scope	Task	Description
FR-17	AI	PreProc.	Each image should be normalised for inference by the AI pipeline.
FR-18	AI	Pipeline	The inference pipeline should be able to run on CPU if no GPU is available.
FR-19	AI	Pipeline	The pipeline should provide appropriate warnings and error messages.
FR-20	AI	Pipeline	The pipeline should find all entities on a plan, extract the text and assign the correct label.
FR-21	AI	Training	The models should be continuously retrained on new user supplied data.
FR-22	AI	Training	Model performance should be stored on the server for monitoring and comparison.
NFR-1	General	Quality	The code should be well structured and follow best practices for each language.
NFR-2	General	Testing	Unit tests should be implemented where appropriate.

Table 1: Requirements

2.4.1 Future requirements

- It was decided not to implement any user authentication or role based access control to focus on other features of the app. If the need arises this could be added at a later point in time.
- Censoring of sensitiv image regions could be easely added in the editor based on the bounding boxes feature. It could also be integrated into the AI pipeline but this would require bigger changes as well as training data that is currently not available.

2.5 User Stories

User stories were used in this project to ensure a user-centric approach in developing the new workflow and app. The usability of the app is of crucial importance because the archive employees spend a lot of time with it. The user stories provide a clear and structured framework for addressing real-world challenges and requirements. They help in translating the technical requirements into actionable tasks, and ensuring that the app is both functional and user-friendly.

2.5.1 Project Management

As an archive employee, I want to create new projects, specifying details such as the project code, architect's name, working directory, and export directory so that I can organise the digitisation tasks efficiently. I should also be able to edit these details later or delete a project.

Risks:
- Incomplete or incorrect project details could lead to major problems with data consistency and difficulty in locating files.
- Accidental deletion of a project could lead to confusion but generally no data is lost.

2.5.2 Project Selection

As an archive employee, I want to see a list of all projects so that I can easily select and continue working on them.

Risk:
- The welcome screen might become cluttered with too many projects, making it difficult to navigate. This can be solved by only displaying the most recent projects and hiding older ones behind a button.

2.5.3 Image Upload

As an archive employee, I want to have full control of the upload process. I want to be able to see the current upload status (uploading / stopped), the server connection, and any files with errors. I want to be able to manually retry a failed upload. Images without a project should never be uploaded. The upload should run continuously in the background and automatically query the upload dirctory for new images.

Risks:
- Interruptions in the upload process could lead to incomplete data transfer and potential data loss. This can be solved by atomic upload and move operations.
- If implemented wrongly continuous background uploads could consume significant bandwidth and system resources, affecting other tasks.
- Intransparent upload errors could lead to confusion, data loss and workflow distruptions.

2.5.4 Visual Editor

As an archive employee, I want a visual editor where I can draw bounding boxes, edit labels and the text of entities, and adjust the image so that I can ensure all metadata is accurate. I want to be able to move the image around and zoom in and out with a fluid framerate.

Risks:
- Errors in bounding box drawing or metadata editing could compromise the quality and accuracy of the digitised data.
- An unresponsive or unintuitive editor could frustrate users, leading to lower productivity.

2.5.5 Nondestructive Editing and Autosave

As an archive employee, I want all my changes to be saved automatically and I want to be sure that all changes can be reverted.

Risks:
- Failure during saving could lead to inconsistend or lost data.
- If edits are performed destructive, the image could be rendered unusable.

2.5.6 Editing Continuity

As an archive employee, I want the app to remember the last edited image when reopening a project so that I can continue from where I left off.

Risks:
- If the app fails to remember the last edited image, users might waste time finding their previous working point or accidentally skip some images.

2.5.7 Documentation

As an archive employee, I want the app to be well-documented so that I and other new users can quickly learn how to use it with minimal guidance.

Risks:
- Poor or outdated documentation could lead to confusion and increased onboarding time. Because the turnover of the people working on digitisation is high, this can tax the archive supervisor excessively.

2.5.8 Oversight and Supervision

As the archives supervisor, I want to be able to check in on the people working on the digitisation from anywhere. I want to be able to setup projects and check the progress.

Risks:
- If the enviorenment synchronisation does not work properly, supervision would become impossible without beeing physically at the photo-station.

3 Architecture & Technologies

3.1 Constraints

The archive provides windows notebooks to the employees working on digitisation. The app needs to run on this hardware, meaning that it does not have access to a GPU.

The archive is currently in the process of migrating their server infrastructure to new hardware with a more powerfull CPU but also no GPU. This server is currently used to run the archive website and the Anton database [2].

3.2 Inference on Device

The initial idea was to integrate the AI pipeline directly into the app. This approach would have the advantage of eliminating the need for server infrastructure. Additionally, the entire project could have been developed using a single language: Python. To test this idea, I built a rudimentary Python GUI app using PyQt [3], integrated the existing machine learning code from the SA project, and attempted to build and package it with setuptools [4].

The first major issue encountered was that PyTorch, the machine learning library used, needed to be installed and built for a specific platform and sometimes even for a specific CPU generation. This requirement made it nearly impossible to build the app once for each platform and then distribute it universally. Consequently, I decided to abandon the approach of performing inference on the device.

3.3 Standalone App

Decoupling the app from the AI pipeline allows for the selection of technologies best suited for each task. Several cross-platform desktop app development platforms exist, with Electron.js being the most popular. Electron.js is an open-source framework developed by GitHub that allows developers to build cross-platform desktop applications using web technologies like HTML, CSS, and JavaScript [5]. By leveraging the Chromium rendering engine and the Node.js runtime, Electron enables the creation of apps that function consistently on Windows, macOS, and Linux. This integration allows developers to use web development technologies to create rich desktop applications, incorporating features like native menus, notifications, and file system access. Popular applications like Visual Studio Code and Slack are built with Electron.

However, Electron apps have long been criticised for excessive memory use and the necessity to bundle a Node.js runtime with the app. The main drawback for this project is the restriction to use JavaScript only. The app's image editing features require computationally intensive matrix operations that need to run very fast, which is difficult to achieve with JavaScript. Therefore, other frameworks were also evaluated [6].

3.3.1 Crossplatform Framework Comparison

An ideal framework would use familiar technologies as well as a fast programming language. The following different frameworks where compared:

3.3.1.1 Flutter

Flutter is an open-source UI toolkit by Google that enables developers to build natively compiled applications for mobile, web, and desktop from a single codebase. Using the Dart programming language, Flutter offers a rich set of pre-designed widgets and tools to create highly responsive and visually appealing user interfaces. It is known for its fast performance and flexible design capabilities, making it well-suited for the project. Although I have some experience with Flutter from the Distributed Systems course where we built a mobile app with Flutter, I am not very familiar with the Dart programming language and the framework as a whole.

3.3.1.2 React Native

React Native is an open-source framework by Facebook that allows developers to build mobile applications for iOS and Android using JavaScript and React. It enables the creation of native-like apps with a single codebase, offering a wide range of components and tools to ensure performance and a smooth user experience. While I have a good understanding of web technologies, I have never worked with React before. Moreover, React Native, like Flutter, is more geared towards mobile app development than desktop apps.

3.3.1.3 Ionic

Ionic is an open-source framework that enables developers to build cross-platform mobile, web, and desktop applications using web technologies like HTML, CSS, and JavaScript. It offers a library of pre-built UI components and tools, allowing for a seamless development process and native-like performance across various platforms. However, Ionic suffers from a similar problem to Electron with the restriction to use less performant JavaScript.

3.3.1.4 .NET MAUI

.NET MAUI (Multi-platform App UI) is an open-source framework by Microsoft designed for building native cross-platform applications for mobile, desktop, and web from a single codebase. Utilising C# and XAML, .NET MAUI provides a unified API and a rich set of UI controls, enabling developers to create responsive and high-performance applications that run on Windows, macOS, iOS, and Android. My main problem with .NET is my unfamiliarity with C# and the Windows development ecosystem. Other than that, MAUI would have been a great fit for the project.

3.3.1.5 Tauri

Tauri is an open-source framework designed for building tiny, fast, and secure desktop applications using web technologies like HTML, CSS, and JavaScript. Unlike other frameworks, Tauri leverages the native operating system's webview, resulting in smaller application sizes and improved performance. This eliminates the need to ship a whole Node.js runtime like Electron. Tauri also offers the ability to leverage the Rust programming language, which I am already familiar with, for performance-critical parts. It provides a flex-

ible API for integrating with native functionalities, making it ideal for creating fast and efficient cross-platform applications.

3.3.2 Stack

The main reasons for choosing Tauri were the familiarity with the technologies (HTML, CSS, and TypeScript for the UI and Rust for the backend). The option to write the performance-critical sections in Rust was also a significant advantage. Instead of JavaScript, I use TypeScript, which provides more type safety. Tauri allows the use of a frontend JavaScript framework, and for this project, I chose SvelteKit due to my familiarity with the technology. The combination of SvelteKit with Tauri is philosophically sound, as SvelteKit, unlike React or Vue, also involves a compilation step.

3.4 Server

The server architecture is relatively straightforward. Python was the obvious choice due to its large ecosystem for machine learning and image processing tasks. I use the OpenCV package for image processing tasks [7]. Implementing a Python web server to make the API accessible was logical since all other server code is already in Python. I chose FastAPI [8] to handle the web traffic because it provides an easy-to-understand API and is lightweight.

For the pretrained models, I made extensive use of the Hugging Face transformers library [9]. This open-source library provides tools and models for natural language processing (NLP), enabling developers to build, train, and deploy state-of-the-art machine learning models for tasks like text generation, translation, and named entity recognition. Hugging Face also offers a platform for sharing and collaborating on models, making advanced NLP accessible. All pretrained models are sourced from the Hugging Face model hub.

To simplify deployment on the ASLA server and ensure a consistent development environment, the entire server is dockerised.

4 App

4.1 User research

Over the last year, I had the opportunity to work on digitisation myself for several weeks and talk to various other employees. Often, the people working for the archive come from Swiss mandatory civil service (Zivildienst) or are interns from OST or college students (Kantonsschule) during a gap year. These employees do not necessarily have a technical or architectural background.

From my own experiences and informal interviews, I gathered the following pain points with the current workflow:

1. Employees need to switch between the photo station, camera app, Excel, and Photoshop frequently, which is time-consuming and disrupts their workflow.
2. The current image viewer is cumbersome and unreliable, making it difficult to locate and read relevant texts on the images.
3. The manual and repetitive nature of the tasks often results in inconsistencies and mistakes, affecting the overall quality and efficiency of the digitisation process.

4.2 Design

Based on the requirements, user stories and insights from the current workflow, a basic wireframe for the app was created in Figma. The three most important screens were blocked out to understand the necessary elements to fulfill the user stories and determine their optimal placement.

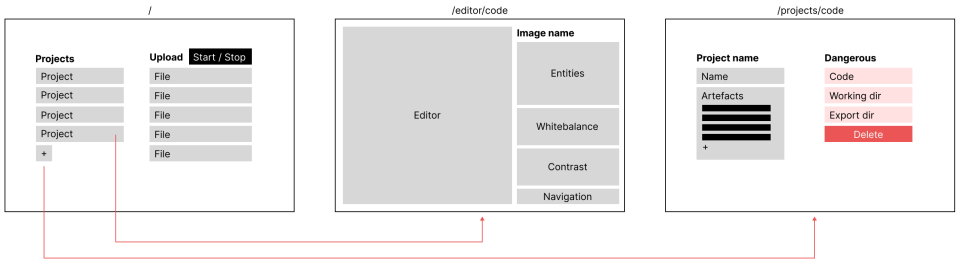

Figure 2: App Wireframes

Next, the wireframes were expanded into mockups. A colour palette was chosen that primarily uses grey values to keep the focus on the content. A desaturated shade of red (#EB5757) is used as an accent colour for certain buttons and interactive elements. The platform's default sans-serif typeface is used: San Francisco on macOS and Segoe on Windows.

Because the UI is built with web technologies, it is fully responsive, with elements expanding and rearranging based on the available space. A minimum window size of 500

by 300 pixels is defined to avoid handling edge cases. This ensures the app remains functional across different screen sizes and resolutions.

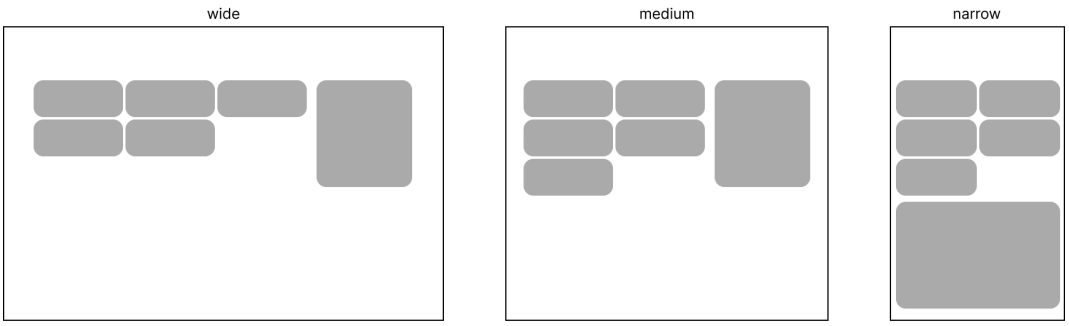

Figure 3: Responsive design

4.3 Structure

The Tauri app is structured into the following components:

4.3.1 Frontend

The frontend is built with SvelteKit, a powerful framework designed for creating high-performance web applications using Svelte. Instead of a virtual DOM, Svelte uses a compiler to generate native JavaScript code that updates only the changed elements. For use with Tauri, all sites are prerendered and then rendered by the platform's native webview: WKWebView on macOS, WebView2 on Windows, and WebKitGTK or WRY on Linux. Tauri ensures that the frontend is sandboxed for security. The UI is built using HTML, native CSS styling, and TypeScript.

4.3.2 Backend

Tauri apps use Rust for their backend logic. The core system manages all interactions with the operating system and provides the webviews. Tauri provides bindings and APIs to interact with the operating system and access native features. These bindings are written in Rust and are accessible from either JavaScript or Rust. Some of the heavily used APIs include:

- Command API: Allows the frontend to call Rust functions.
- File System API: Provides secure and unrestricted file system access.
- Window API: Manages the different windows of the app.

This structure allows for a clean separation of concerns, ensuring that the frontend can focus on the user interface and interactions, while the backend handles the core logic and system-level operations efficiently.

4.4 Components

4.4.1 Splash Screen

A simple splash screen is shown during app startup. This screen is not clickable and has no window controls, as it is generally only displayed for a second.

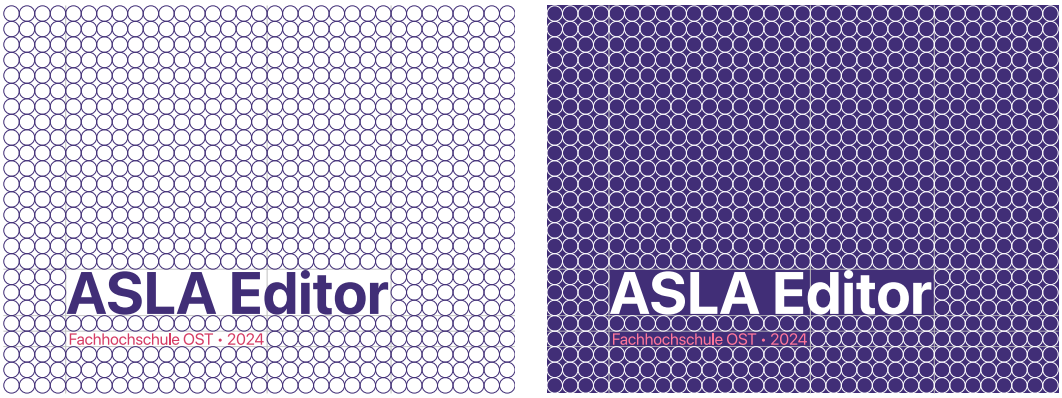

Figure 4: Splash screens for light and dark mode. The design is inspired a project by Ernst Cramer from 1965: Friedhof Uetliberg.

4.4.2 Welcome Screen

The app manages state with different configuration files. The first one is the local app config, located in the app config directory at `os_config_dir/ch.asla.asla-editor/app.config`. This directory is created when the app is installed. In addition to user preferences, this JSON file contains the path to the `environment.config` file. This file should exist only once in the ASLA filesystem. It contains all the project settings, as well as the path to the upload directory and API endpoint.

Because this environment file is shared among all clients, changes to it are reflected on all running clients. This allows multiple users, such as an archive supervisor and an archival intern, to work simultaneously. By creating multiple environment files, multiple instances can be managed, for example, an older archived environment or a test and development environment. If the path to the environment file is not valid or present in the app config, the user is prompted to specify or create a new one.

```
{
  "projects": [
    {
      "code": "KLA",
      "name": "Klauser Gebrüder",
      "workingDirectory": "/.../projects/KLA/working",
      "exportDirectory": "/.../projects/KLA/export",
      "subfolders": true,
      "artefacts": [
        "Klauser",
        "KLA",
        "ETH SIA BSA"
      ]
    }, ...
  ],
  "uploadDirectory": "/.../upload",
  "apiEndpoint": "http://localhost:30500/image/"
}
```

Listing 1: Environment config file

4.4.3 Homescreen

The home screen of the app contains a list of the most recent projects on the left and the upload manager on the right, as well as a settings button.

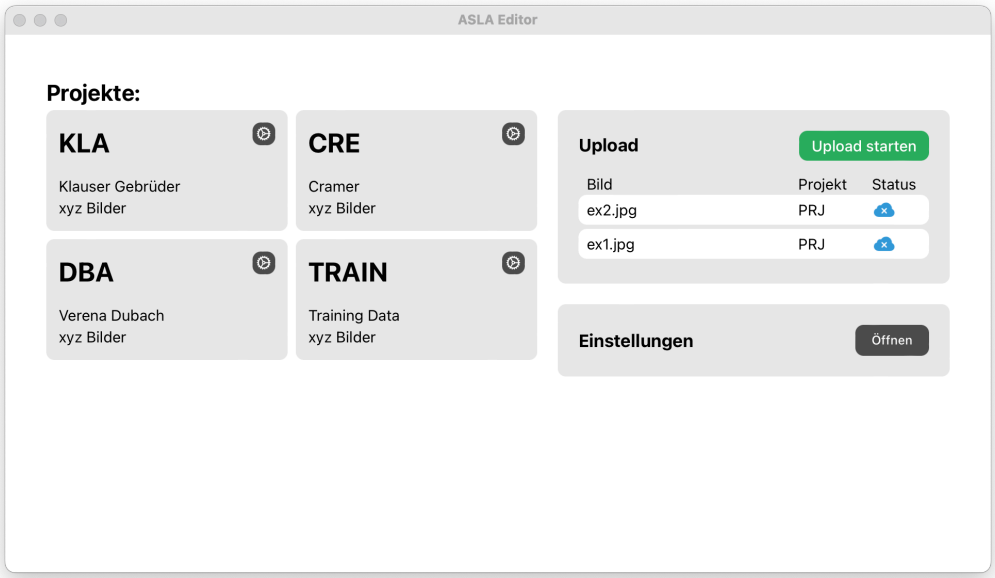

Figure 5: Homescreen view

4.4.4 Upload

The upload component handles the image upload process. It consists of two parts: the upload Svelte component for the UI and the upload service singleton class. The user can start and stop the upload and view the status of the images in the upload directory. The component loads the API URL from the environment config file and reads all the images in the upload directory. The directory is queried every 2 seconds, and all the image paths are stored in a writable array [10] using Svelte stores, to which the UI subscribes. If the server is not reachable, the component displays a warning banner at the top with a refresh button, and the start button is disabled. If an image upload fails, the image is marked, and a retry button is shown, allowing the image to be manually sent to the server.

If the server successfully predicts the metadata for an image, the upload component receives it. According to the filename, the image and the metadata are moved to the corresponding working directory. This operation is atomic, ensuring that if one write fails, the initial status is restored, maintaining data integrity. Because this operation requires arbitrary file system access, it is handled by the Rust code. The upload service class calls the `process_image` function via a Tauri command:

```
try {
  const prediction: ApiResponse = await invoke('process_image',
    {path: image.path, name: image.name, endpoint: STATE.apiEndpoint}
  )
} catch (err) {
  this.failedImages.push(image.name)
}
```

Listing 2: Call the `process_image` Rust function from JavaScript (shortened)

The upload component is built from the ground up with robustness in mind. It can handle network errors, inference errors, and file system errors without crashing or losing data.

4.4.5 Project

Clicking on the settings icon in the top right corner of a project leads to the project screen. Here, all the project settings can be changed. The screen is split into a regular section on the left and a dangerous section on the right.

ID	Scope
Name	Display name of the project, functionaly irrelevant.
Artefacts	List of tokens that should be ignored during named entity recognition. This helps the AI avoid misclassifing the name of the architect as the name of the client and so on.
Code	This string is used to match an image to this project, eg: ASLA_KLA_100305_1.jpg
Working directory	The path where the images and predictions from the AI Pipeline are saved to.
Export directory	The path where a copy of the images are saved to during exporting.
Structure	Sets a project up for nested directories (not implemented)

Table 2: Project settings

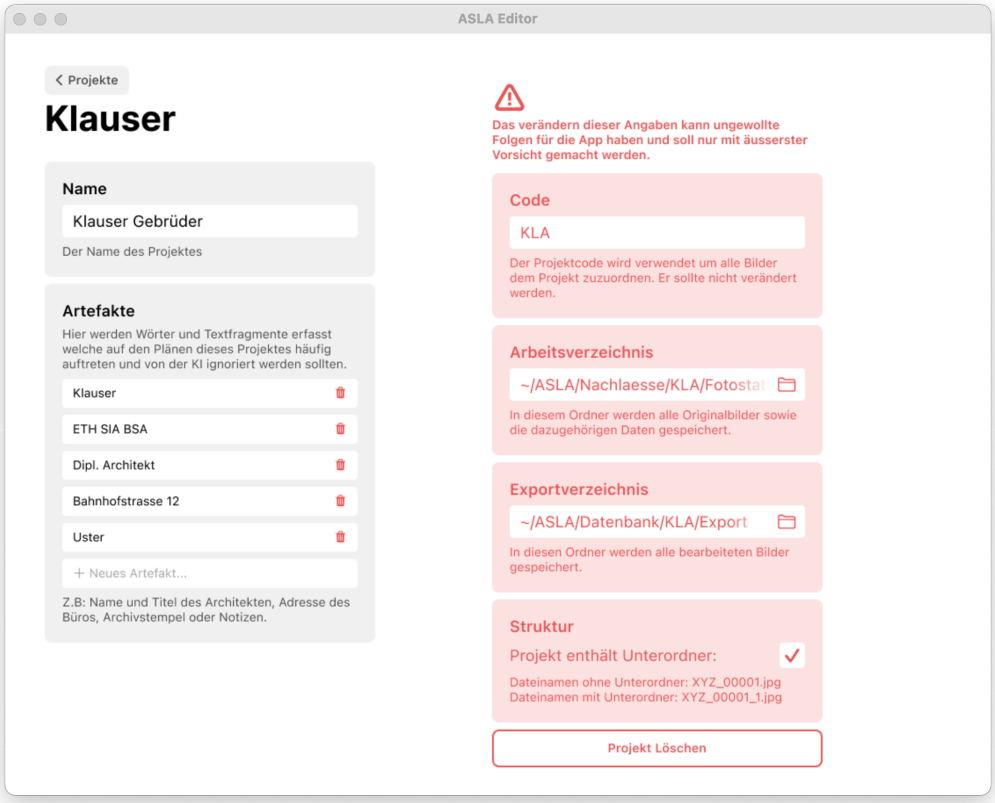

Figure 6: Project view

4.4.6 Editor

The largest and most complex component is the Editor. Since users will spend the most time here, it is crucial that the user experience is well thought out, sophisticated, and

reliable. The Editor consists of the Svelte editor component with its sub-components and the Rust backend to handle the image data.

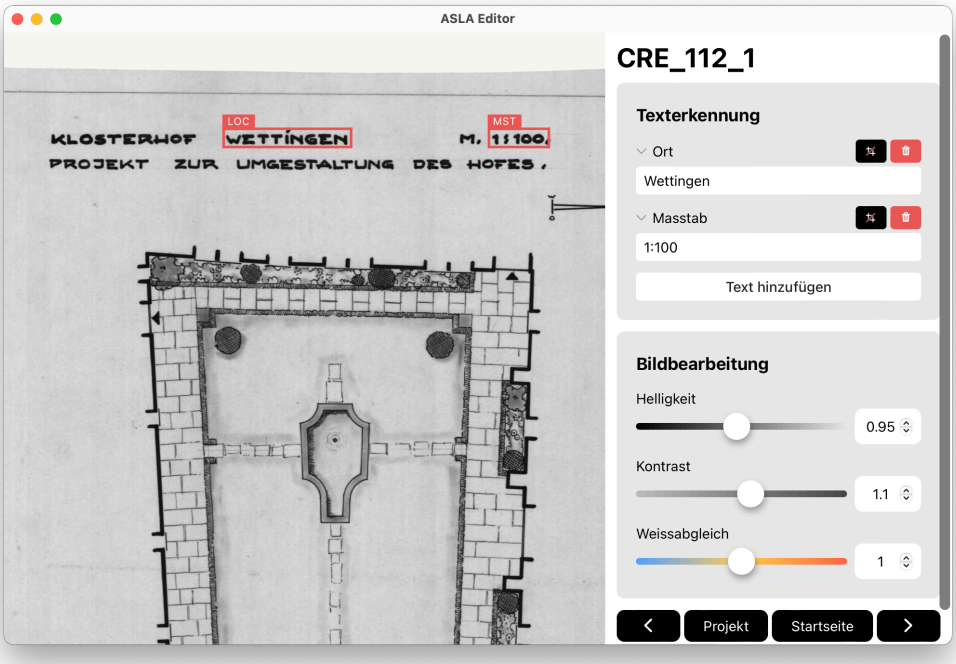

Figure 7: Editor view

4.4.6.1 Image selection

When the Editor is opened, the correct image needs to be loaded. Every project stores the name of the last opened file in a hidden file called `.current` inside its working directory. If this image does not exist anymore or if it's the first time the project is opened, the first image is selected.

4.4.6.2 Rust Backend

The backend creates an `Editor` struct when the app is launched. This struct allocates memory to hold the image data and stores the current image path. If the frontend requests image data, the Rust code reads the image based on the provided path. The image is read as an `ImageBuffer` that stores the RGB pixel values as `Vec<u8>` with three entries. If the path is the same as the current image, the image is not reloaded but served from the current buffer. Memory access is controlled with a `MutexGuard`.

If the user edits the image's white balance, contrast, or brightness, the image data needs to be updated. Since real-time performance is crucial for user feedback, the pixel-level updates are performed in parallel with Rayon:

```rust
image
  .rows_mut()  // creates an itarator over the mutable rows of this
image
  .par_bridge()  // parallelises the iterator: one row per thread
  .for_each(|row| {
    row.for_each(|pixel| {
      *pixel = update_pixel(*pixel, white_balance, contrast, brightness)
    })
  });
```

Listing 3: Image update function in Rust (simplified)

The `update_pixel` function implements a custom update function that combines all three operations, white balance, contrast and brightness into a single calculation:

$$f(p_i) = c * (p_i * w_i - 128) + b + 128 \mid p \in \{(p_1, p_2, p_3) \mid 0 \leq p_i \leq 255\}$$

Figure 8: Pixel update function, where w: whitebalance, c: contrast and b: brightness

The current implementation needs to send the image data as a base64 encoded string to the frontend. This encoding step results in several hundred milliseconds of processing time, which does not allow for real-time image editing. The ideal solution would be to have a shared memory block between the front and backend. However, this feature is not currently available in Tauri but is expected to be included in the next version [11] with changes to the inter-process communication.

The Rust backend also writes and reads the metadata json files. Each image has an identically named `.json` file with the following structure:

```
{
  "entities": [
    {
      "label": "loc",
      "text": "Aarau",
      "hasBoundingBox": true,
        "boundingBox": { "top": 940, "right": 2571, "bottom": 996,
"left": 2427 },
      "manuallyChanged": false
    } ...
  ],
  "grading": {
    "contrast": 0.95,
    "brightness": 1.1,
    "whiteBalance": [1, 1, 1],
    "manuallyChanged": true
  },
  "format": {
    "crop": { "top": 197, "right": 1574, "bottom": 258, "left": 1603 },
    "rotation": 0.014,
    "manuallyChanged": false
  }
}
```

Listing 4: Metadata file

4.4.6.3 Viewport

The left two thirds of the editor are dedicated to the current open plan. The viewport gets the image data from the rust backend and displays it with an HTML `img` element. The editor has two main interactive functions: Navigation and drawing.

Navigation consists of move and zoom. Both are implemented with a transformation matrix:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s & 0 & tx \\ 0 & s & ty \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Figure 9: Transformation, where s: zoom level and tx: translation

This transformation matrix is updated on every mouse move or wheel event and updates the position and scale of the image accordingly.

The user can drag the plan in any direction with the mouse. Translation is implemented with a mouse move listener that provides the `moveX` and `moveY` properties which are mapped to tx and ty in the transformation matrix. Moving the plan out of the viewport is prevented with a min and max value depending on the zoom level.

The user can change the scale of the plan with the mouse wheel. The scale is limitied to a range between 0.5 (50%, zoomed out) and 5 (500%, zoomed in). After every mouse wheel

event the scale is calculated and used in the transformation matrix as s (the same value for the x and y axis). This scales the image from the center which is not what most users expect. More intuitiv is that the zoom origin is at the current pointer position. This is implemented by calculating the position of the mouse pointer after applying the transformation matrix and then correcting the x and y translation for this change:

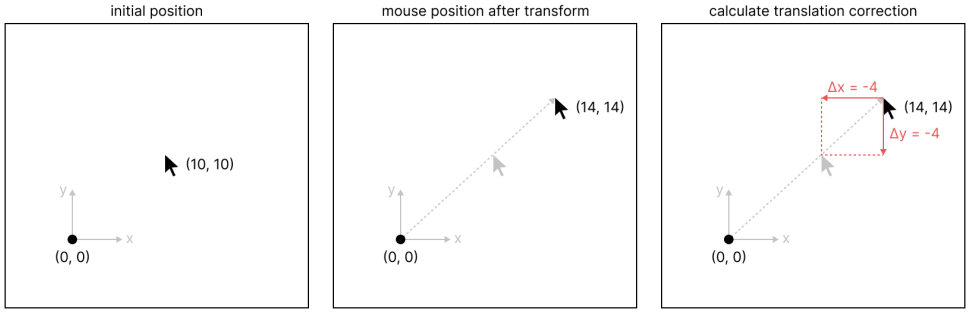

Figure 10: Scale from mouse pointer

The navigation allows the user to inspect the plan in detail and to read small text. It also allows the plan to be positioned so that the bounding boxes for the entities can be inspected and edited. A new bounding box can be started from an entity in the controls panel. In drawing mode, mouse drags do not map to translation anymore but instead span a bounding box. The same `moveX` and `moveY` event properties are used to calculate the size of the bounding box. An existing bounding box can be resized from any corner. Each corner has an hoverable 25 pixel big region which displays a drag handle if the mouse is inside it.

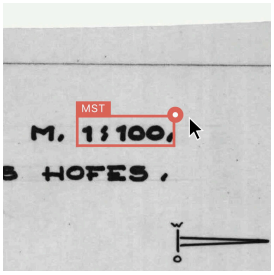

Figure 11: Bounding box drag handle

4.4.6.4 Controls

On the right side of the editor are the controls. This sidepanel is scrollable. On the top is the name of the current image displayed. The first panel below that is the text detection (Texterkennung). Every entity is displayed with its label (drop down), a button to remove or create the bounding box, a delete button to remove the entity and an input containing the entitiy text. A new entity can be created by clicking on a button at the bottom of the list. When a new entity is created, the viewport automatically switches to the drawing mode to create the bounding box, and the input is focused for streamlined data input. When the user hovers over an entity the corresponding bounding box (if present) has a red drop shadow to help find it in the viewport. This should avoid time spent searching for the text on the plans, something that is quite common in the current workflow.

The next panel containts the image editing settings. It currently contains three slider for brightness, contrast and whitebalance. If one of these values changes, the Rust `update_image` function is called to update the pixel values of the image.

4.4.6.5 Shortcuts

Keyboard shortcuts are a useful tool to make the user experience more seamless. The following shortcuts are implemented for the editor:

`CMD` + `0` : Reset the viewport, center it, and scale it to 100%

`CMD` + `n` : Create a new entity, equivalent to a button click

`ESC` : Cancel drawing operation

`<` / `>` : Previous / next image

4.5 Buildsteps

Because the app needs to support different platforms, an automated build step is implemented. Tauri requires the build process to be run on the target architecture because it depends on native libraries. This is implemented with a GitHub Action:

1. Set the build environment, e.g., macOS or Windows.
2. Check out the repository with the source code.
3. Install Node.js (version 20) for the frontend.
4. Install the Rust toolchain (stable) for the backend.
5. Install npm packages: `npm install`
6. Build the app: `npm run tauri build`
7. Upload the executable.

The final executable can then be distributed (via website, for example) and installed on the target system.

4.6 Testing & Bugfixes

During the development of the AI pipeline, I needed to gather training data. Naturally, I used the app to create entities and draw bounding boxes. This process proved extremely valuable as I spent considerable time with the editor. I identified what worked well and the small pain points that became disruptive when repeated dozens of times. For example, the initial size of the dragging area was too small, resulting in many aborted entity resizes. During this period, I found several bugs and issues that had escaped during development.

This testing of the app was especially useful because, at that time, no one was working for the archive, so I couldn't let them test the app.

4.7 User Guide and Documentation

The functionality of the app, as well as the entire workflow, is documented in the user guide (Appendix). In the five-page document, every step of the digitisation process is explained. The document is written in German and aims to reduce the training overhead for new employees. Since the users are often at the archive for only a short amount of time, the clear instructions should help streamline the onboarding process and free up time for the supervisors.

5 Introduction to Transformer-Based Deep Learning

In recent years, the field of artificial intelligence has witnessed remarkable advancements, particularly in the domains of Natural Language Processing (NLP) and Optical Character Recognition (OCR). At the forefront of these advancements is the development and application of transformer-based models. This chapter aims to provide the necessary conext for the implementation of the AI Pipeline with a focus on OCR and Named Entity Recognition (NER).

5.1 Transformer Models

Transformer models, introduced by Vaswani et al. in 2017 [12], revolutionized the field of NLP by addressing the limitations of recurrent neural networks (RNNs) and convolutional neural networks (CNNs). The primary innovation of transformers is the self-attention mechanism, which allows models to weigh the importance of different words in a sentence regardless of their position. This architecture enables parallelisation, reducing training times and improving the handling of long-range dependencies.

Transformers consist of an encoder-decoder structure. The encoder processes the input sequence (can be text or image) and produces a continuous representation, which the decoder then uses to generate the output sequence. Key components of the transformer architecture include multi-head self-attention, position-wise feed-forward networks, and positional encoding.

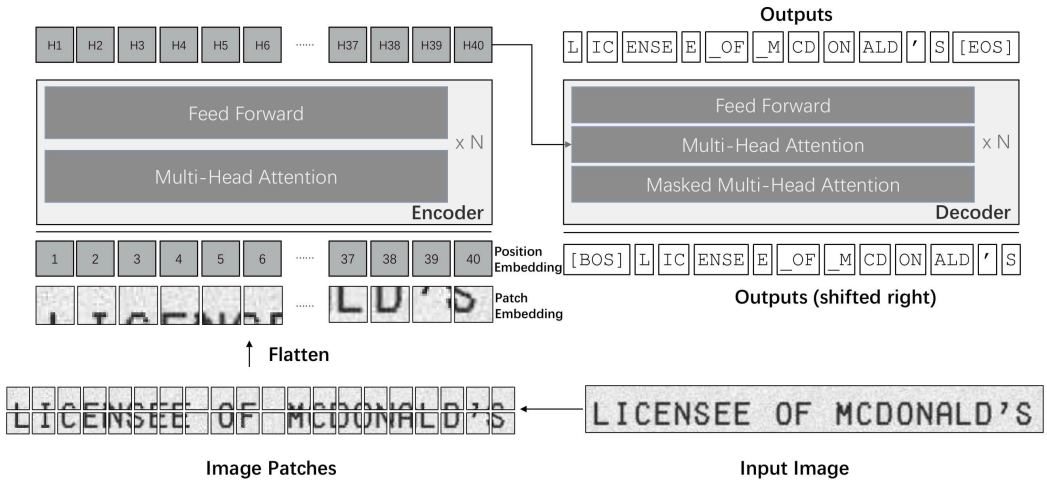

Figure 12: TrOCR Architecture [13]

5.1.1 Encoder

The encoder's primary function is to process the input sequence and transform it into a continuous representation. It comprises a stack of identical layers, each containing two main sub-layers:

1. Multi-Head Self-Attention Mechanism: This mechanism allows the encoder to attend to different positions within the input sequence simultaneously. By computing attention scores, the model determines the relevance of each word in the sequence relative to every other word. The multi-head aspect refers to having multiple attention heads that can focus on different parts of the sequence independently, capturing various aspects of the relationships between words.
2. Position-Wise Feed-Forward Networks: Following the self-attention mechanism, each position in the sequence is passed through a fully connected feed-forward network. This network consists of two linear transformations with a ReLU activation in between. The feed-forward network is applied independently and identically to each position, contributing to the model's ability to handle long-range dependencies.
3. Positional Encoding: Unlike RNNs, transformers do not inherently capture the order of tokens in a sequence. Therefore positional encoding is introduced to provide this information. Positional encodings are added to the input embeddings and incorporate information about the position of each token in the sequence. These encodings can be learned or predefined, allowing the model to distinguish between different positions.

5.1.2 Decoder

The decoder is responsible for generating the output sequence based on the continuous representation produced by the encoder. Like the encoder, the decoder is also composed of a stack of identical layers, but with an additional sub-layer to handle the encoder-decoder interactions:

1. Masked Multi-Head Self-Attention Mechanism: The decoder uses a masked version of the multi-head self-attention mechanism to ensure that predictions for a given position can depend only on known outputs up to that position. This masking prevents the model from attending to future tokens in the sequence.
2. Encoder-Decoder Attention Mechanism: This sub-layer allows the decoder to attend to the entire encoded input sequence. It operates similarly to the self-attention mechanism but focuses on the encoder's output rather than the decoder's own previous outputs. This enables the decoder to incorporate information from the input sequence into its generation process.
3. Position-Wise Feed-Forward Networks: As in the encoder, each position in the decoder sequence is passed through a feed-forward network to process the attended information and produce the final representation.

4. Positional Encoding: Positional encodings are also added to the input embeddings of the decoder to provide information about the position of each token in the output sequence.

5.1.3 Workflow

The overall workflow of the encoder-decoder structure in a transformer model can be summarized into the follwing steps:

1. Input Processing: The input sequence is first embedded and combined with positional encodings.
2. Encoding: The encoder processes the input embeddings through multiple layers of self-attention and feed-forward networks, producing a continuous representation.
3. Decoding: The decoder generates the output sequence by attending to the encoder's continuous representation and its own previously generated tokens, applying masked self-attention, encoder-decoder attention, and feed-forward networks at each layer.
4. Output Generation: The final layer of the decoder produces the output sequence, which is then transformed (typically through a linear layer and softmax function) into probabilities over the target vocabulary to generate the final predictions.

5.2 OCR with Transformers

OCR is the process of converting different types of documents, such as scanned paper documents, PDFs, or images captured by a digital camera, into editable and searchable data. Traditional OCR systems rely on pattern recognition and machine learning techniques. However, transformer-based models have shown significant promise in enhancing OCR accuracy and efficiency.

The application of transformers in OCR involves the use of vision transformers (ViTs) and text recognition transformers. ViTs apply the transformer architecture to image patches, treating them as sequences, similar to words in a sentence. This approach has been effective in capturing the spatial dependencies within images. Furthermore, text recognition transformers combine the principles of vision transformers and traditional sequence-to-sequence models to decode text from images.

5.3 NER with Transformers

NER is a crucial task in NLP that involves identifying and classifying named entities (e.g. names or locations) within a text. The advent of transformer models, particularly Bidirectional Encoder Representations from Transformers (BERT), has significantly improved NER performance.

BERT and its variants (like German-BERT or RoBERTa) leverage the transformer architecture's bidirectional self-attention mechanism, enabling the model to consider the context of a word from both its left and right sides. This contextual understanding is crucial for accurately identifying named entities. For instance, the word "Apple" can refer to a

fruit or a technology company, depending on the context. Transformer-based models effectively disambiguate such cases by understanding the surrounding context.

6 AI Pipeline

6.1 Foundation

During my SA project, *"AI for Digitisation of Historic Architectural Plans"*, I explored different approaches and deep learning models to find an effective image processing pipeline for the given problem. I proposed a three-model architecture, which forms the basis for this thesis. This chapter will explain the setup, with all the work done during the SA clearly marked. The existing parts of the pipeline are summarized here, readers interested in more details can find a detailed description in the SA paper.

Additionally, even before the SA, I created a CNN-based marker detection script. This approach was deemed too elaborate and resource-intensive, as marker detection could be achieved without deep learning by using template matching. Reimplementing marker detection is part of this BA.

6.1.1 Learnings from the SA

The most relevant conclusion from the SA was that the models needed training data from the archive. Synthetic and simulated training data could only improve accuracy to a certain point. Additionally, to train the layout and OCR models, bounding boxes are needed, which did not exist. Based on this, I decided to work with the models as they were and implement them in a way that would allow automatic and continuous retraining on the data gathered daily by archive employees using the new app. This approach saved me from spending days and weeks gathering training data myself.

Another insight from the SA was that the NER model had difficulty distinguishing the architect's name from a client's name. This issue arises because both are names. The same problem occurred with the address of the architect's office, which is often present on the plans. To solve this problem, I created a list of artefacts for each project. This list can be filled out when a new project is started. The artefacts are sent with every image and used by the NER model to ignore these predictions.

The final insight from the SA was the importance of preprocessing every image before it is handed to the machine learning models. I believed that this technique could improve the models' performance, as other papers [14], [15] recommended it.

6.1.2 Task

The main task of the AI pipeline is to find and name the text entities on the plan. But it also needs to automate the following steps, that are currently done manually:

1. Crop and rotate the image correctly
2. Remove the archive stamp
3. Automatic retraining and model changes
4. Resource management
5. Performance evaluation

6.1.3 Architecture

With the app in place, the server needs to handle the uploaded images and send a prediction back. It consists of the following components:

1. A FastAPI webserver to handle the uploaded images
2. Marker detection and orientation
3. Stamp removal
4. Preprocessing
5. Text localisation and clustering
6. Text extraction
7. Named entitiy recognition
8. Postprocessing

And for the retraining:

1. API to recive training images from the app
2. Retrain models
3. Evaluate new models and compare to the current ones
4. Update the better performing models

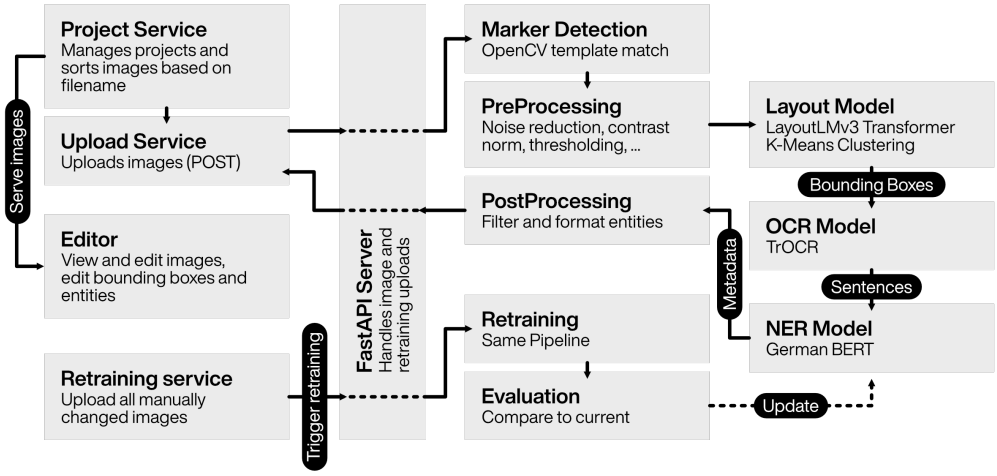

Figure 13: Architecture Diagram

6.2 Evaluation

One of the major pain points in the SA was the lack of an evaluation or test suite for the different models. Although I attempted to formalize and automate the evaluation process, the diverse architectures and outputs of the models required many tests to be performed manually. For this thesis, I developed a `test_model` function that automates the evaluation process. This function allowed me to verify that each modification and addition to the pipeline actually improved model performance.

The test function automates the evaluation of model performance by running the provided models with a fixed set of test images and metadata. After each test, the results are saved into the benchmark directory and a dictionary with the results is returned.

```
BENCHMARK REPORT

---------------- Config
layout model: microsoft/layoutlmv3-large
ocr model:    ../models/ocr/ocr_v3
ner model:    ../models/ner/ner
test images:  11

---------------- Results
layout model: 0.267
ocr model:    0.971
ner model:    0.425

---------------- Comments
Testing ocr model, trained with 10 epochs, 77 training samples
```

Listing 5: Benchmark file example

With the test_model function, I could continuously verify improvements in model performance. For instance, after implementing stamp removal or new preprocessing steps, I reran the test suite to compare the results with previous benchmarks. This automated testing approach ensured that each change to the pipeline was validated, leading to more consistent improvements. Approaches that did not work could be stopped early.

6.3 API

To handle image uploads I implemented a very basic webserver with FastAPI [16]. It exposes the followin routes:

`/ping`	`GET`	Used by the app to check if the server is online
`/models`	`GET`	Returns the currently loaded models with their evaluation score
`/image`	`POST`	Takes an image and a list of artefacts and returns a prediction
`/retraing`	`POST`	Takes an image and its metadata and stores it in the retraining directory

6.4 Model Service

To enable dynamic management of the models and their memory usage, I created a model service class that manages the models. This class is instantiated once when the server starts and abstracts access to the models. It exposes a `run_pipeline` method that performs inference and can load or unload models based on a specified path. The currently used models are saved in the project root in a `ai_config.json` file. This file defines the the currently used models and the corresponding test score. Additionally the file defines the retraining settings like number of images and training iterations.

The first time the pipeline is run, the models are loaded, and a timer is set (currently to 30 minutes). Each time the pipeline is run (i.e., a prediction is requested), the timer is reset.

If the timer runs out, all models are unloaded to free up shared system resources. This also allows the retrained and improved models to be loaded automatically.

6.5 Marker detection

The first step currently done manually by the archive employee after an image is taken is cropping and rotating it so that the plan is correctly aligned. Previously, I created a Python script that used a convolutional neural network (CNN) to detect four magnets with high-contrast patterns. For this thesis, I developed a new and simpler method using template matching. This approach is feasible because the markers always have the same distance to the camera and therefore the same size.

Figure 14: The image used for template matching

1. A template of the marker is iterated over the entire image, and the matching score for each position is calculated using convolution.
2. Due to the larger size of the marker, the pixels around the center have high values, making it impossible to simply take the four highest values, as they could all correspond to the same marker.
3. To address this, we use another kernel (twice as big) to stride over the image and select only the highest value at each step.
4. Finally, the selected values are ordered and filtered one last time for closeness to ensure that two pixels from the same marker where not coincidentally at the border of the kernel.
5. The pixel coordinates of the four highest values are returned as the marker positions.

This improved method runs much faster than the old CNN-based approach while maintaining similarly high accuracy. The streamlined process ensures efficient and accurate alignment of the plans, enhancing the overall digitisation workflow.

6.6 Preprocessing

One way to improve model performance is by editing the images to make text detection easier. When done manually, the results are very good. By adjusting the contrast (gradient curve), denoising, and sharpening, we can improve model performance by more than 20%. The challenge is that each image requires different adjustments.

My initial approach was to implement something similar to the manual process. I defined a set of image heuristics that would then be corrected based on their values. For example, an image with low contrast would have its contrast increased. However, this method did not improve the models' results.

Metric	Calculation
Contrast	Michelson contrast
Noise	Standard deviation between the original and a smoothed image
Brightness	Mean value of all pixels
Sharpness	Laplacian Variance
Entropy	Shannon Entropy

Table 3: Image metrics

Next, I explored more advanced image processing techniques, which typically focus on extracting the edges of letters. Two of the most successful pipelines were as follows:

Preprocessing 1

1. Convert image to grayscale.
2. Enhance contrast with CLAHE (Contrast Limited Adaptive Histogram Equalisation).
3. Reduce noise with a Gaussian blur.
4. Binarize the image with a Gaussian adaptive threshold.
5. Apply a morphological close operation with a 3x3 kernel.
6. Detect edges with Canny edge detection, using thresholds of 50 and 150.

Preprocessing 2

1. Convert the image to grayscale.
2. Enhance contrast with a weighted function.
3. Binarize the image with an inverted binary threshold.
4. Apply a distance transform with a mask size of 5.
5. Normalize the image.
6. Threshold again with OTSU's method.
7. Apply a morphological close operation with a 7x7 kernel.

Both preprocessing pipelines worked well for certain types of images but rendered others completely unusable.

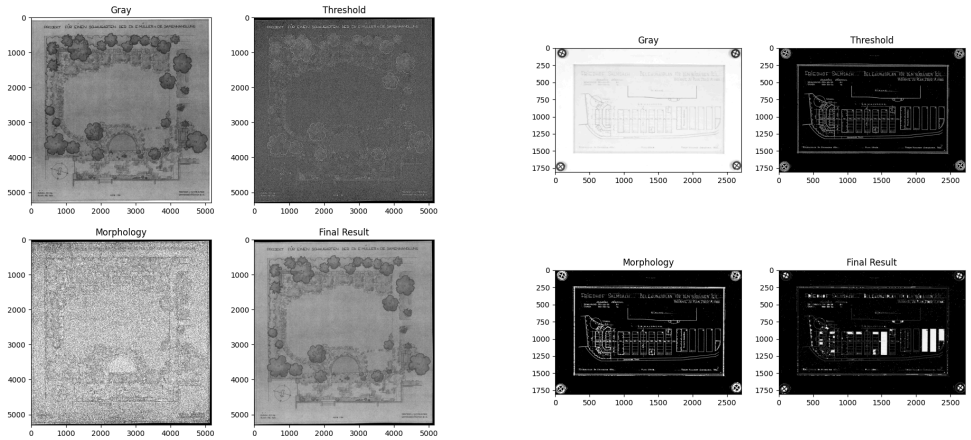

Figure 15: The same preprocessing performed on two images. The text on the left image is more readable and the noise is significantly reduced. The text on the right image has some closed characters and is much less readable.

Because the plans span a wide period and were created by different people in vastly different styles, I could not find a preprocessing approach that works for all images. One potential solution is to generate several different images and run them all through the AI pipeline, selecting the one with the most plausible predictions. However, this approach is unfeasible because the inference must be run without a GPU, and predictions already take 30 seconds per image. To cover a reasonable set of configurations, each operation would have to generate several images, resulting in dozens of generated images from a single plan.

My final approach, which is now implemented, involves taking the edge and contour enhancing techniques and applying them as a mask to the image. This results in a much less drastic alteration while making the preprocessing more general and robust for different images. The effect on the models is only marginal with a improved accuracy of about 2.5%

This approach balances the need for enhancing image features relevant to text detection while preserving the overall integrity of the original image. By using the edge and contour enhancements as a mask, important features are highlighted without overly modifying the image, thus maintaining a consistent preprocessing pipeline suitable for the diverse range of images.

```
  adjustment # The enhanced image, similar to the previous pipelines
  mask # image with only 0 or 1 pixels

  adjustment = 255 * factor
  enhanced = np.where(
     mask == 1,
     np.clip(image + adjustment, 0, 255),
     np.clip(image - adjustment, 0, 255)
  ).astype(np.uint8)
```

Listing 6: Applying the enhanced image with a contour mask

6.6.1 Stamp removal

(This code was reused from the SA) Some plans within the dataset contain a stamp, which was introduced during the archival digitisation process.

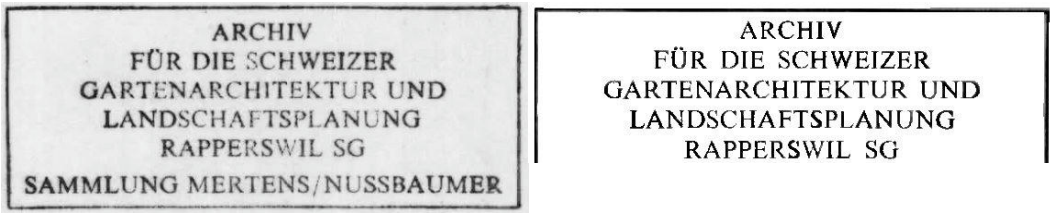

Figure 16: Example of a stamp on the left and edited template on the right with the bottom row containing the estate cut off.

The text present on these stamps is detected and extracted by the layout and OCR model. However, this extraction process can lead to confusion for the NER model and result in longer inference times. To address these issues, a template matching algorithm is employed before the image is processed by any model, similar to the marker detection. If the algorithm identifies coordinates above a specified confidence threshold, it samples and averages the colors of six surrounding pixels to determine the background color of the plan in that region. Subsequently, the entire stamp is replaced with this color, effectively eliminating it from the plan without any sharp lines or artifacts that could influence the text detection. Tests have shown that a modified version of the stamp where the bottom row and line are cut off works the most reliably for the template matching algorithm.

6.7 Layout Model

The task of the layout model is to find the bounding boxes for all words on a plan. These bounding boxes are important for archive employees to quickly locate relevant text on a plan. Additionally, the OCR model that follows the layout model is limited to single-line inputs, meaning it cannot process images with multiple lines of text, hence the need to locate each word by its own.

In the SA project, I proposed the LayoutLM architecture, developed by Xu et al. [17], designed for document image understanding. For this thesis, I upgraded the base model

to version three [18]. This upgrade required only minor changes to the code since the architecture of version three is very similar to version two. The new model performs 4% better than its predecessor.

6.7.1 Clustering

(This code was reused from the SA) On architectural plans, words are organized into groups, such as the plan header, creation date and location block and the architect's address and signature, among others. These groupings play a pivotal role in named entity recognition as they provide essential contextual information. Consequently, there is a need to spatially cluster the words to preserve their relationships. To achieve this, a k-means algorithm is employed, given its capability to group related words based on their proximity. As the actual number of groups is unknown, the algorithm is executed with varying k values ranging from 2 to 12. The k-value with the highest silhouette score is then selected as the optimal clustering parameter. Finally, the boxes in each cluster are ordered from top to bottom, left to right based on their coordinates. This approach ensures a robust grouping mechanism for enhancing the accuracy of named entity prediction.

6.8 OCR Model

After the layout model has located the words and clustered them into groups, the OCR model is used to extract the text from each subimage.

(From the SA) The TrOCR model [19] proposed in the SA consists of a vision transformer (encoder) and a language transformer (decoder). The encoder divides and flattens the input image into a single row of patches and then generates image embeddings. The decoder takes these embeddings and produces the string output. Both the encoder and decoder consist of multi-head atten- tion and feed-forward blocks. The decoder additionally has a masked multi-head attention layer. This architecture lends itself to extensive pretraining, and is therefore ideal for our use case where training data is not yet available. TrOCR has several pretrained models. The large-handwritten is the most accurate model but also the slowest. One problem is that TrOCR needs a single line input image, meaning it can't find words on a page or do predictions on multiline text blocks. Therefore, this model can only be used in collaboration with another model, like LayoutLM.

1. The image is cropped for each bounding box detected by the layout model. Each cropped image contains a single word or a group of words that need to be recognized.
2. The cropped images are passed to the OCR model, which predicts the text within each bounding box.
3. The predicted text from the OCR model is cleaned up by removing special characters and filtering out single-character predictions.
4. The cleaned text is annotated with a confidence score for the post-processing step. This score helps in deciding which predictions to keep if several entities of the same type are predicted during NER.

6.9 NER Model

The final model in the AI pipeline is the named entity recognition model.

(From the SA) Named Entity Recognition is a natural language processing (NLP) technique, focused on identifying and classifying entities within textual data. Classically these entities range from persons and organisations to locations, dates, numbers, and more. Text is typically tokenized, meaning broken down into smaller units called tokens, which can be words or subwords. These tokens are then classified into predefined categories. This classification is based on the context and characteristics of the token. NER models are often based on transformers or RNN encoders.

The following entities are relevant for the archive:

Entity	**Label**	**Description**	**Examples**
Client	CLT	The person or organisation owning the property	Hr. Dr. Müller
Location	LOC	The location of the project, could be street, municipality, region or a combination	Bahnhofstrasse 12, Rapperswil
Scale	MST	The scale of the plans	1:50, 1:100, 1:250
Date	DATE	The creation date of the plan	1.3.1941, 54
Unlabeled	O	All words without a label	des, und, Baum

Table 4: List of entities

Preliminary tests in the SA showed that existing models do not perform very well on the data, with an accuracy of all models in the single-digit percentages. An off- the-shelf model would also not work for the given use case because it requires several custom entities. The client entity for example is a compound of a name, with title, and organisations. This necessitates training on custom data. In the SA this was done with manually gathered and synthetically generated data. The final model that was trained during the SA was a BERT variant for the german language [20], [21].

The NER model recives the predicted words from the OCR model and combines them into a sentence based on the clustering done by the Layout model. For each sentence the model predicts a possible entitiy per word.

6.10 Postprocessing

Before returning the entities back to the app, a postprocessing step is applied.

1. Empty or unlabeled entities get filtered out
2. Entities matching a project artefact, for example the same name as the architect, are filtered out
3. Scale entities are formatted into scale format '$X : Y$' if possible
4. If several entities have the same label, the ones with the highest confidence score are chosen.

6.11 Comparison to LLM's

During the AI pipeline selection in the SA, the potential use of large language models (LLM) was discussed. Due to the classification of some of the archive's data as sensitive personal information, sending data to external services like ChatGPT is not possible. Nonetheless, to compare the results with state-of-the-art LLMs, I selected some images without personal information and queried the OpenAI API with the following question:

```
Kannst du mir bitte folgende text-daten aus dem Bild herauslesen:
- Titel des Projektes / Dokumentes
- Ortschaft wo das Projekt umgesetzt wurde
- Bauherr oder Bauherrin des Objekts
- Massstab des Plans, nur die beiden Zahlen also z.B. 1:100, 1:10 oder 5:1
- Datum des Plans formatiert als DD.MM.YYYY
Bitte gib mir die informationen als JSON mit folgenden keys:
{
    "titel": Titel des Projektes,
    "loc": Ortschaft,
    "clt": Bauherr,
    "mst": Massstab,
    "date": Datum,
}
Verzichte bitte auf jeglichen Text ausser dem JSON sowie auf formatierung
mit Newlines usw.
```

I tested the models GPT-4 Turbo and GPT-4 Omni. Overall, ChatGPT performed very well in image extraction. It could format dates and scales consistently and return data in a structured manner. While it did make mistakes, it generally performed better than the pipeline from the SA. However, it could not return accurate bounding boxes for the words. Although it provided pixel values, these were always incorrect.

I also tested Cohere's Command R+ and Meta's LLaMA-3 models, but neither could provide a result.

If, in the future, an open-source large language model becomes available that can run on hardware provided by the archive, replacing the current AI pipeline with an LLM could be a valid option. If the model's accuracy is sufficiently high, the bounding boxes might even be omitted, given that fewer corrections would be needed and retraining might not be required anymore.

7 Retraining

7.1 Proof of model improvement

As discussed in Section 6.1.1: "Learnings from the SA", my focus for this thesis was on the pipeline and retraining rather than gathering extensive training data. However, to demonstrate that additional training data improves model performance, some data collection was necessary. I manually annotated 80 images in two sets.

While the inference code for the three models remains largely unchanged from the SA, the training pipeline for the layout and OCR models was created during this thesis. In the SA, the only model that was trained was the NER model.

1. The first and larger set is the training set. These images are prepared exactly as an archive employee would with the app, just without any AI assistance. Each image in the training set has a metadata file containing a list of all entities with bounding boxes.
2. The second set is the test or evaluation set. In addition to the metadata file, each image in this set has a text file containing the sentences for the NER model. This is necessary because the metadata only contains the entities without the context of the surrounding words, e.g., "Garten des **Herrn Gretsch** in **Rapperswil**".

The training code for the models is similar to the inference code but involves more preprocessing steps. For example, when creating a dataset, the images cannot be inlined into the encoding because this leads to memory overflows due to the large image sizes. Instead, I created a custom dataset class, based on the Torch dataset class, that only loads and returns an image when it is needed during training.

```
class ImageDataset(TorchDataset):
  def __init__(self, hf_dataset, processor: Processor):
    self.dataset = hf_dataset
    self.processor = processor

  def __len__(self):
    return len(self.dataset)

  def __getitem__(self, idx):
    item = self.dataset[idx]
    image = cv2.imread(item['image'])
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    encoded_inputs = self.processor(
      images=image,
      text=item['tokens'],
      boxes=item['bboxes'],
      word_labels=[label2id[label] for label in item['ner_tags']],
      padding="max_length",
      truncation=True,
      return_tensors="pt",
    )
    for k, v in encoded_inputs.items():
      encoded_inputs[k] = v.squeeze()

    encoded_inputs['bbox'] = encoded_inputs['bbox'].to(torch.int64)

    return encoded_inputs
```

Listing 7: Custom Dataset Class that can handle large images

With this training data, I trained the models first on half the images and then on all images.

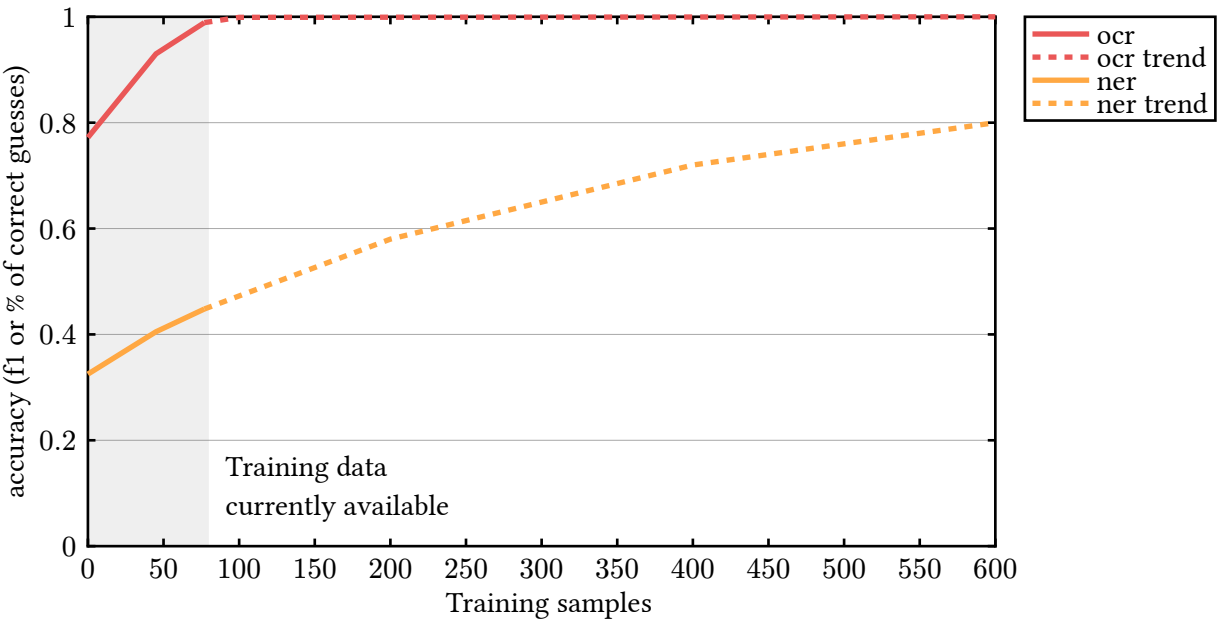

Figure 17: Model evolution

The accuracy of the OCR model could be improved to more than 98% with the little training data that was gathered meaning that further retraining will only bring marginal improvements. The accuracy of the NER model could be improved about 10% from the base model trained during the SA and currently reaches an accuracy of 46%. This is more then twice as good as random guesses (5 entities = 20%), but still not very accurate. If we calculate the improvement that one additional training image provides we get 0.13%. If we extrapolate this improvement per image with more training data, also taking a decay of accuracy gains into account, we can predict that after 600 images, a model accuracy of 80% can be reached. This amount of training data can be gathered by an archive employee in less than a month, thanks to the new app.

Based on these results, I am confident that focusing on the retraining pipeline is the right approach for this thesis.

7.2 Training

As mentioned in Section 7.1 “Proof of model improvement”, the training code needed to be implemented for the layout and OCR Models. I used the PyTorch machine learning framework and ressources from the web, most notably the Huggingface Hub.

7.2.1 Training Data

The app records all entities that are changed manually with a flag in the corresponding metadata json file. These images and metadata are uploaded to the `/retraining/` endpoint. A handler saves the images in the retraining directory on the server.

The data service class provides access to the training data. When data is requested for a training run, a fixed number of images are selected. The images are randomly picked from the directory but their likelyhood is determined by their date, meaning that newer images

are picked with a higher propability than older ones. This should prevent a training run with images from one physical role or folder of plans. Plans from one source like this can all be from the same project and therefore be very similar.

7.2.2 Schedule

Retraining is scheduled to happen every saturday morning at 02:00 when the server should have the least amount of traffic to the asla website. The retraining function is run by a `BackgroundScheduler` with a `CronTrigger` from the `apscheduler` python package.

7.2.3 Retraining

Retraining runs the retraining function from all models with the training data selected and loaded by the data service. The retrained models are named according to the convention model_name_vX, where the version number "X" is automatically incremented based on the most recent model. The models are initialised with the wheights of the currently active model.

After training a model it is evaluated. A new instance of the AI service class is created and loaded with the retrained models. The `test_model` method evalutes the new models and the result is compared to the current model score. Each model that has an improved score is replaced. The model path is saved in the `ai_config` file and automatically loaded the next time the ai service loads the models.

7.2.4 Harware limitations

Training the NER model works well on a CPU, with an average epoch taking less than 5 minutes per 100 images. This makes it feasible to retrain the NER model on existing hardware. However, the OCR model requires GPU training. I used OST's high-performance computer, the DGX-2, equipped with 16 Tesla V100 graphics cards, each with 32GB of shared GPU memory and 1.5TB of system memory. The combined GPU memory of 512GB and over 2 petaFLOPS of computational power made this setup more than powerfull enough for my models. Training the OCR model required only one Tesla card but I needed to reduce the batch size from 16 to 8 in order to prevent out of memory errors. The following configuration was used:

Argument	Value
Number of training epochs	100
predict_with_generate	True
Evaluation Strategy	Steps
Per Device Batch Size	8
Fp16 (Mixed precision)	True
Number of Dataloaders	4
Gradient Accumulation Steps	4
Max Token Length	64
Early Stopping	True
No Repeat N-Grams	3
Length Penalty	3
Number of beams	8

Table 5: OCR Training arguments

All training runs were tracked using Weights and Biases (W&B) [22]. W&B is a comprehensive machine learning platform that facilitates the tracking, visualisation, and optimisation of machine learning experiments. It offers tools for experiment tracking, hyperparameter tuning, and model versioning. Integrating seamlessly with PyTorch, it provides real-time insights into various metrics during training. I used W&B extensively during the SA project, and it proved invaluable during this thesis by allowing me to fine-tune the training process, such as setting the optimal number of steps.

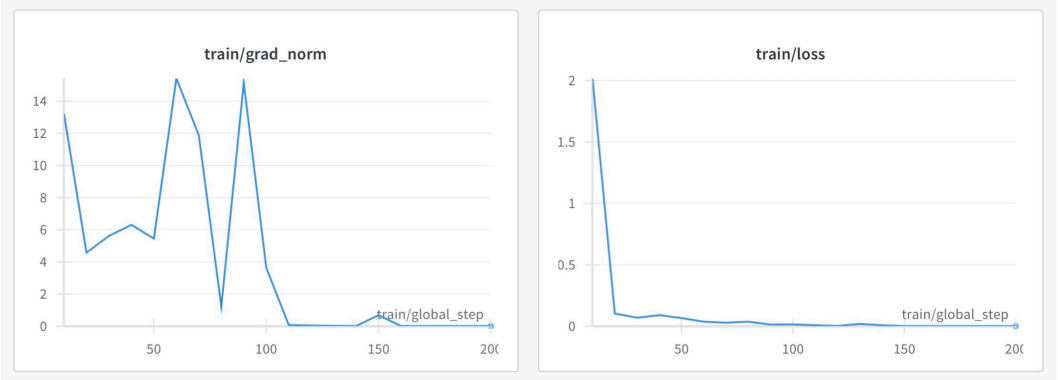

Figure 18: OCR Training loss of the model v4 (current best)

Given that the OCR accuracy is already above 98%, further retraining is not necessary, eliminating the need for GPU resources for OCR retraining.

The Layout model, while trainable on a CPU, takes much longer than the NER model. A training run with 100 images would take several days on a CPU compared to minutes on the DGX-2. Therefore, I conducted my training using the graphics cards. Continuous retraining of the layout model is probably not feasible with the current setup.

This leaves two options: either upgrade the ASLA server with a GPU or run the backend on a different machine, or alternatively, perform the layout retraining manually at periodic intervals.

7.2.4.1 DGX-2 and Apptainer

Apptainer [23] is a container platform similar to Docker but tailored for scientific and high-performance computing (HPC) applications. It ensures reproducibility and portability across different computing environments, from local development to large-scale HPC systems. Apptainer is the required method to run code on the DGX-2. I created a custom Apptainer image for training the OCR and Layout models, along with scripts to upload training data and download the trained models. With this infrastructure in place and monitoring with W&B, retraining the models periodically should be a simpler process, though it does not match the efficiency of an automatic workflow.

8 Conclusion

8.1 New Workflow

The new workflow with the app and the AI pipeline is as follows:

1. The ASLA Editor desktop app is opened, and the upload process is started.
2. The employee selects a new roll or folder of plans.
3. The first plan is positioned on the photostation, and the corners are secured with the marker magnets.
4. The image is taken.
5. The plan is labeled with a unique name (e.g., ASLA_KLA_100305_1) and the image is automatically or manually named with the same identifier.
6. After the roll or folder is photographed, all images are moved to the upload directory.
7. The current project is opened in the app, displaying the last edited image.
8. The employee clicks through the images, verifying the text of the entities. The bounding boxes help them quickly locate the entities.
9. If an entity is not correct, it is quickly corrected.
10. At the end of the day, the app is closed without the need to save anything.
11. When the employee returns on Monday, the models have learned from the manual corrections and are hopefully a bit better.

8.2 BA Task definition

This thesis builds upon the foundation laid by the SA "Digitise Historic Architectural Plans with OCR and NER Transformer Models." In the SA the primary objective was to develop an AI pipeline capable of detecting handwritten text on historical plans, performing Optical Character Recognition (OCR) to identify words, and subsequently extracting a set of named entities (NER). The focus of this Bachelor's thesis is to leverage the insights and apply the proposed models to create a desktop application for use by the archive. The desktop application is designed to improve and speed up the archival process. The images from the camera are transmitted to a server, where they are processed by the AI pipeline. The results are sent back to the application where the user can refine the AI predictions as needed, enhancing the overall efficiency and accuracy of the digitisation process.

8.3 Evaluation

The following assignments where defined at the start of this Bachelors thesis:

8.3.1 Build an app that can be used by the archive employee:

Task: **Required**
Define functional and non-functional requirements based on the needs of the archive
Result
Both types of requirements where defined based on interviews with the archive, especially Simon Orga.

Task:	**Required**
Write user stories based on the needs of the archive	
Result	
Different user stories where written based on the requirements.	

Task:	**Required**
Build a desktop GUI app that runs on the archives computer	
Result	
The app runs cross platform on macOS and Windows (Linux is not tested) thanks to the chosen framework and the buildpipeline.	

Task:	**Required**
Import the images from the camera and send them to the server	
Result	
The app's upload manager can upload images from a directory and handle any errors. The user can control the upload as well as select images manually.	

Task:	**Required**
Receive the processed images from the server and display them with the prediction results	
Result	
The upload manager also handles the returned predictions and saves them in the working directory for that project. The Editor can read the prediction and display the image with the bounding boxes and entities.	

Task:	**Required**
The user can edit or override the predicted metadata	
Result	
The predicted text can be edited by the user.	

Task:	**Optional**
The user can delete, edit or draw new bounding boxes for possible future model training	
Result	
The bounding boxes can be created, deleted and edited by the user in the visual editor. Shortcuts for the most important actions are available.	

Task:	**Optional**
Add basic image editing capabilities (whitebalance, exposure, contrast) to manually tweak the settings	
Result	
Whitebalance, exposure and contrast can all be changed in the editor. One minor downside is that rendering the image takes a few hundred milliseconds, therefore adjsutments are not quite real time.	

8.3.2 Deploy the AI pipeline on a server

Task:	**Required**
Containerise the AI pipeline and make it accessible via REST API (send image, recive prediction)	
Result	
The pipeline is dockerised and deployed with a FastAPI webserver.	

Task:	**Required**
Add preprocessing to the AI pipeline: • Marker detection and cropping • Image enhancements with whitebalance, exposure, contrast • *Dynamic* image processing to make the image better suited for the AI pipeline like denoising, thresholding, distance transforms and opening morphological operations	
Result	
Marker detection is implemented and works. Preprocessing is implemented but does not meaningfully improve model	

Task:	**Required**
Build a test suite that measures the pipeline performance to make changes to it measurable and comparable	
Result	
The test suite can benchmark an arbitrary combination of models and save the result as text and json.	

Task:	**Required**
Try to improve the performance of the specific models and pipeline as a whole	
Result	
The performance of the pipeline could only be improved marginally. The new pre- and post-processing steps did not bring big accuracy increases. But the retraining pipeline could lead to significant improvements.	

Task:	**Required**
Compare inference performance on different server hardware (CPU / GPU)	
Result	
The inference and especially the test performance was evaluated.	

Task:	**Optional**
Implement continuos retraining and model improvement with new data	
Result	
Retraining is fully implemented, including automatic model updates. Retraining the layout model proved to be difficult on CPU.	

Task:	**Optional**
Visualise model evolution	
Result	
Visualising the model scores is not implemented. The data is available it only needs some kind of UI to visualise it.	

Task:	**Optional**
Image stiching for plans that do not fit in a single photograph	
Result	
This is not implemented and would require bigger changes in the upload code as well as changes in the server code.	

8.3.3 Deliverables

The following deliverables where specified:

- Desktop app
- Containerised server app
- Pipeline test suite
- Complete source code and research
- User guide and documentation for archive employees

8.3.4 Requirements

All functional and non-functional requirements specified in Section 2.4 have been achieved.

8.3.5 Achievements

With the creation of the app and the implementation of a functional AI pipeline, this thesis can be considered a success. The app alone will significantly enhance the efficiency and quality of the digitisation process, as well as improve the daily workflow of archive employees. The app has clear business value for the archive.

Currently, the AI pipeline is only partially useful, depending on the specific plan being processed. However, this thesis has laid the groundwork for the improvement of prediction quality and accuracy over time. Achieving an F1-score of 80% within a few months would be transformative, reducing the digitisation timeline by years.

8.3.6 Code Base

Although the code base was rarely highlighted in this thesis, a significant amount of time and effort was dedicated to its development. The code is meticulously crafted and adheres to modern programming standards. Many components were rewritten several times to incorporate new ideas, insights, and learnings, enhancing their functionality and reliability.

The architecture of the app and the AI pipeline is highly modular, allowing individual components to be exchanged or updated independently. For instance, any of the three models can be replaced by entirely different ones, provided their predictions conform to the expected data structure. Additionally, a substantial portion of the functionality is covered by unit tests, ensuring robustness and reliability.

Overall, the thoughtful design and rigorous development of the code base contribute significantly to the project's success, facilitating future improvements and adaptations.

8.4 Future Work

The app as well as the AI pipeline can be extended in the future. Especially the model evolution needs to be monitored. As new open source models are developed, they can be tested and compared to the current ones. Especially the capabilities of large language models could be interesting and well suited for the problem. The app can be extended with the ability to handle very large plans by automatically stiching them together.

Another interesting project could be the incoorperation of redaction. Certain plans contain sensitiv information that needs to be redacted before the plans are published on the web. In a first step, the app could be equipped with the ability to let the user redact parts. This data could be gathered over months of normal work until enough training data is gathered, so that a machine learning model could be trained to propose which parts to automatically redact.

Finally the export process could be directly integrated into the publishing workflow so that the archive supervisor could export and upload the images to the web with a single click.

11 Appendix

11.1.1 Disclaimer: Advanced Tools

In the process of writing this thesis I used different tools:
SemanticScholar was used to summarise and find literature, provided by a machine learning model.
ChatGPT was used to correct grammar mistakes and edit some passages for clarity. It also helped me find problems to coding challenges and helped with the Typst syntax.
JetBrains AI was used during developement. It provides advanced code completion.

11.1.2 Documents

- Agreement of own contribution
- Copyright agreement
- Publication agreement
- BA Task Definition
- Time planning

List of Abbreviations

AI Artificial Intelligence
API Application Programming Interface
ASLA Archive of Swiss Landscape Architecture
BA Bachelor of Arts
BERT Bidirectional Encoder Representations from Transformers
CLAHE Contrast Limited Adaptive Histogram Equalisation
CNN Convolutional Neural Network
CPU Central Processing Unit
DGX Deep Learning System by NVIDIA
GPU Graphics Processing Unit
HPC High-Performance Computing
JSON JavaScript Object Notation
LLM Large Language Model
NER Named Entity Recognition
NLP Natural Language Processing
OCR Optical Character Recognition
OST Eastern Switzerland University of Applied Sciences
RNN Recurrent Neural Network
SA Semesterarbeit (Semester Project)
W&B Weights and Biases

List of Figures

List of Listings

List of Tables

Bibliography

[1] ASLA, “Archiv für Schweizer Landschafts Architektur.” 2024.

[2] K. & Ritter, “Anton, The archive database.” [Online]. Available: https://www.anton.ch/

[3] Q. for Python Development Team, “Qt for Python.” 2023.

[4] P. P. Authority, “setuptools.” 2023.

[5] E. Community, “Electron.” 2023.

[6] JetBrains, “Cross-Platform Frameworks.” 2023.

[7] O. team, “opencv-python.” 2023.

[8] S. Ramírez, “FastAPI.” GitHub, 2018.

[9] H. Face, “transformers.” 2023.

[10] Svelte, “Svelte Stores.” 2024.

[11] FabianLars, “Tauri V2 IPC.” 2024.

[12] A. Vaswani *et al.*, “Attention is all you need,” in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[13] M. Li *et al.*, “TrOCR: Transformer-based Optical Character Recognition with Pre-trained Models.” 2022.

[14] W. Bieniecki, S. Grabowski, and W. Rozenberg, “Image Preprocessing for Improving OCR Accuracy,” in *2007 International Conference on Perspective Technologies and Methods in MEMS Design*, 2007, pp. 75–80. doi: 10.1109/MEMSTECH.2007.4283429.

[15] M. R. Gupta, N. P. Jacobson, and E. K. Garcia, “OCR binarization and image preprocessing for searching historical documents,” *Pattern Recognition*, vol. 40, no. 2, pp. 389–397, 2007.

[16] S. Ramírez, “FastAPI.” Python Software Foundation, 2018.

[17] Y. Xu *et al.*, “LayoutLMv2: Multi-modal Pre-training for Visually-Rich Document Understanding.” 2022.

[18] Y. Huang, T. Lv, L. Cui, Y. Lu, and F. Wei, “LayoutLMv3: Pre-training for Document AI with Unified Text and Image Masking.” 2022.

[19] M. Li *et al.*, “TrOCR: Transformer-based Optical Character Recognition with Pre-trained Models.” 2022.

[20] B. Chan, T. Möller, M. Pietsch, and T. Soni, “Open Sourcing German BERT Model.” [Online]. Available: https://www.deepset.ai/german-bert

[21] T. M. Branden Chan Stefan Schweter, “German's Next Language Model,” 2020.

[22] Weights and B. Inc., “Weights and Biases: Experiment Tracking for Machine Learning,” *Weights and Biases Documentation*, 2024, [Online]. Available: https://wandb.ai/

[23] A. Community, “Apptainer: Application containers for scientific and high-performance computing,” *Apptainer Documentation*, 2024, [Online]. Available: https://apptainer.org/