Zurich University
of Applied Sciences

School of
Engineering
CAI Centre for
Artificial Intelligence

Bachelor Thesis Computer Science

Digitalization of Chess Scorecards

Author	Nico Ambrosini
	Gian Hellinger
Main Supervisor	Prof. Dr. Mark Cieliebak
Assistant Supervisor	Pius von Däniken
Date	17.06.2022

DECLARATION OF ORIGINALITY

Bachelor’s Thesis at the School of Engineering

By submitting this Bachelor’s thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Bachelor thesis have been partially or wholly taken from other texts and represented as the student’s own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor’s and Master’s Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date:	**Name Student:**
Winterthur, 17.06.2022	Nico Ambrosini
Winterthur, 17.06.2022	Gian Hellinger

Abstract

Chess is one of the best-known board games in the world. It is played in private for fun, the open streets of large cities, organized clubs, and competitive and prestigious tournaments offering high financial rewards. And wherever a large community comes together to enjoy a shared passion, there are opportunities to be seized in peripheral services.

During competitive chess matches, the players record their moves on scorecards. These are used to verify the course of the match, but also to analyze it later using chess software. Usually, they need to be manually replayed on a virtual board. This thesis aimed to improve an existing application that allows to digitalize scorecards into a machine-readable file format by using optical character recognition (OCR).

A prototype Python application has already been created in previous works. It transforms handwritten cards into a standard file format by use of external APIs, and forms the foundation of this project. The advances done in this work mostly concern the previously rather rudimentary graphical user interface (GUI), missing multi-user support, and elevating the existing foundation to a production-ready state.

To achieve this, proficient chess players were invited to participate in giving live feedback to the prototype, which was then used to design useful features to make the GUI clearer, more intuitive, and powerful.

A fully-fledged web application was developed as a result, using modern tools and frameworks like Bulma, a CSS framework for a coherent interface design, Docker, a commonly used container virtualization platform to ensure portability, and multiple Javascript chess libraries. The most prominent features are an animated chessboard that automatically plays through the optically recognized moves, and a responsive interface optimized for mobile devices.

According to user feedback, the goal of making a clear, intuitive interface was fulfilled, while production-readiness was achieved with limitations, due to the implementation and deployment lacking a proper security analysis and architecture.

Zusammenfassung

Schach ist eines der bekanntesten Brettspiele der Welt. Begeistert wird es gleichermassen in organisierten Clubs, öffentlichen Plätzen und in prestigeträchtigen, hoch dotierten Turnieren gespielt. Bildet sich eine grosse, diverse Community um eine gemeinsame Leidenschaft, eröffnen sich dann auch immer Möglichkeiten, damit verbundene Dienste zu entwickeln und anzubieten.

Der Verlauf kompetitiver Schach-Matches wird während des Spiels von den Teilnehmenden auf Scorecards niedergeschrieben. Dadurch wird der Ausgang der Partie verifiziert, und sie kann später manuell auf einem virtuellen Schachbrett nachgespielt werden, um Details und begangene Fehler zu analysieren. Diese Arbeit verfolgte das Ziel, eine existierende Applikation zu verbessern, welche den Prozess dieser Digitalisierung in ein maschinenlesbares Format mithilfe optischer Zeichenerkennung (OCR) automatisch durchführt.

Als Basis dient ein Python-Prototyp, welcher in vorangehenden Arbeiten entwickelt wurde. Er nutzt externe APIs grosser Anbieter, um die handgeschriebenen Scorecards in ein standardisiertes Dateiformat umzuwandeln. Die Weiterentwicklungen dieses Projekts betreffen hauptsächlich die bisher eher rudimentäre grafische Benutzerschnittstelle (GUI), die fehlende Mehrbenutzerfähigkeit, und das Erreichen eines veröffentlichbaren Zustandes.

Dazu wurden erfahrene Schachspielerinnen und -spieler eingeladen, an einer Live-Evaluation der Originalversion teilzunehmen. Auf Basis dieses Feedback wurden nützliche Funktionen entwickelt, um das GUI übersichtlicher, intuitiver und mächtiger zu gestalten.

Als Resultat entstand eine vollwertige Web-Applikation, welche moderne Mittel nutzt, wie z.B. Bulma, ein CSS-Framework für ein kohärentes Design, Docker, eine vielgenutzte Container-Virtualisierungs-Plattform um Portabilität zu gewährleisten, und verschiedene Javascript Schach-Libraries. Die auffallendsten Erweiterungen sind ein animiertes Schachbrett, welches automatisch die erkannten Züge durchspielt, und eine für den Gebrauch mit mobilen Geräten optimierte Oberfläche.

Gemäss Nutzerrückmeldungen wurde das Ziel, ein übersichtliches und intuitives GUI zu entwickeln erreicht. Funktional ist eine Veröffentlichung nun ebenfalls möglich, wobei eine zusätzliche Sicherheitsanalyse und -architektur nötig sein könnte, um einen angemessenen Datenschutz gewährleisten zu können.

Preface

The goal of this project was to take an existing application and improve its user experience. This demanded all the knowledge we gained in the past few years at ZHAW and other working experience. For some parts we had to do further research and for others we thought do be doing well, just to realize in the end, that we walked into the same traps so many did before us. We had the chance to see how artificial intelligence and optical character recognition works, although our focus was not on it, and it made us discover the world of chess. And really made us want to dive into it a little more.

Starting from a pre-existing codebase seemed like an advantage in the beginning, but it took as far longer to get a good footing and understanding of the code than expected – not only because of varying levels of code quality and cleanliness throughout all modules, but also because of its structure, the frameworks used which none of us had any experience with, and the fact that many of those do not have a large online community to rely on. But the more we worked on it, the more proficient we became navigating its quirkiness and the more enjoyment we gained from implementing new features and seeing how it all turned out. It is almost a shame, that we had to finish up the project, when we had so many more ideas and could barely stop ourselves from implementing them – which brought us many sleepless nights in the end, since there was still a large part of this report to write, when the time budget got more and more critical. Overall, it was a very fun, fulfilling experience, but also one of the most taxing and stressful periods of our lives so far, considering everything else also going on besides it.

Taking this opportunity we would like to extend our sincere thanks to Prof. Dr. Mark Cieliebak and Pius von Däniken, who guided us through the project, always kindly giving advice and suggestions during our weekly meetings. We would also like to express our deepest appreciation to Gundula Heinatz and Michael Blum for spending their precious free time to test the application and give us amazing feedback and suggestions on improving the application. Their knowledge of chess helped us delivering a product, that can hopefully convince users of its value with intuitiveness and style.

Winterthur, 17.06.2022

Contents

1 Introduction

Analysis, theorizing, and practice are not typically the selling points when trying to advertise sports or games. Nevertheless, they are very crucial activities for any player who aspires to become better in their discipline, be it traditional, mostly physical sports, or more mind- and tactics-focused ones like e-sports and chess.

In terms of chess, this usually includes re-playing and thoroughly thinking about every move made throughout a series of matches, for example at a tournament. To achieve this, players must note down every single move of the game while playing, using a piece of paper – usually a specific form provided for the cause, called scorecard (see 2.1 Chess Scorecards) – and the chess Standard Algebraic Notation (see 2.2 Standard Algebraic Notation (SAN)). After the tournament, the sheets must be manually entered into an online database and analysis tool like ChessBase [1], from where it can then be analyzed directly, or downloaded as a PGN-file (see 2.5 Portable Game Notation (PGN) File Format), to be imported into other chess applications.

This is a tedious process and does not only demand valuable time, but also makes this theoretical approach for refining and improving a person's skill more of a chore than an interesting and fun activity. Especially newer players might be turned away from putting work into honing their skills like this, if it includes a lot of dull, repetitive tasks.

There must be ways to optimize this process to not only take less time to complete, but also be more intuitive and interesting for rookie players.

1.1 Initial Situation

Before the beginning of this work, multiple projects had already been conducted on the topic. As a result, a web-application had been developed, to automize as much of the process of digitalizing scorecards as possible.

Also there existed some competitors on the market, who were aiming for similar goals, but have had significant drawbacks. The prior students had tried to compensate for those, trying to create a unique selling point.

1.1.1 Preceding Works

The original version of the application was a product of the bachelor thesis of Béla Horváth and Colin Dreher [2] in 2020. It was meant to be a proof of concept for a Python-based algorithm, that uses optical and intelligent character recognition (OCR/ICR, see 2.3 Optical Character Recognition) to read all moves of a chess-match from a variety of table-based scorecards, before checking them for correctness interactively and exporting all information as a standard PGN-file.

It was created and deployed successfully, fulfilling the goals set. The focus was to optimize the recognition algorithm to reduce mistakes in reading the hand-written sheets as much as possible, reducing the need for manual inputs. As a suggestion for further improvement, changes to the optical recognition algorithm, extraction of metadata from the sheets, breaking the monolithic approach into smaller constructs, user management, multi-user support, and more, were mentioned.

Later in 2020, the application was revisited in a project work by Albin Abduli [3]. It focused on fixing issues with recognizing characters that are overlapping with lines and other characters. Recommendations for further development mostly stuck with those of the previous work, also mentioning a custom-trained neural network (see 2.4 Convolutional Neural Network) as a substitute for the ABBYY (see 3.3.1 ABBYY) recognition tool.

The last precedent modification was applied as objective of Volkan Caglayan and Bernt Nielsen's bachelor thesis in 2021 [4]. It contained a new, modified user interface, processing of non-table scorecards (still requiring a specific structure) and improvements to character recognition and move correction. Instead of only using a single OCR provider, a variety of services were tested, compared, and combined.

Again, the training of an evolutionary neural network was suggested as a way of improving recognition quality, instead of manual trial-and-error to define parameters used with implemented OCR-providers. To be able to process scorecards that span multiple sheets (and therefore multiple files), using OCR-provider Amazon Rekognition (see 3.3.4 Amazon Rekognition) is mentioned as a possibility. Also, the proposal for multi-user support, better scalability through breaking the monolithic approach and using a database for saving scorecards (and possibly training the recognition algorithm) were reinforced.

1.1.2 Base Application “Very Chess”

The previous work’s product is an application called “Very Chess”, derived from the word “verify”. It resembles the base on which this work expanded and built upon and is also called the “prototype” in this work. The following paragraphs describe the functions and limitations of the application from a user’s perspective.

Workflow

“Very Chess” serves a single purpose and therefore only provides one workflow: the automized transformation of a physical, filled in chess scorecard into a PGN-file, that includes all necessary information about the match played.

The user is greeted with a clean and slick looking logo and the immediate prompt to upload the image of a scorecard and start the process (Figure 2). Selecting (or using drag-and-drop) a valid file and hitting the start-button makes the UI go inactive, displaying a spinner telling the user, that the OCR is currently analyzing the image (Figure 1).

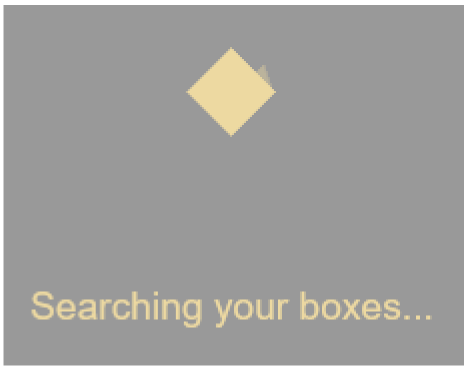
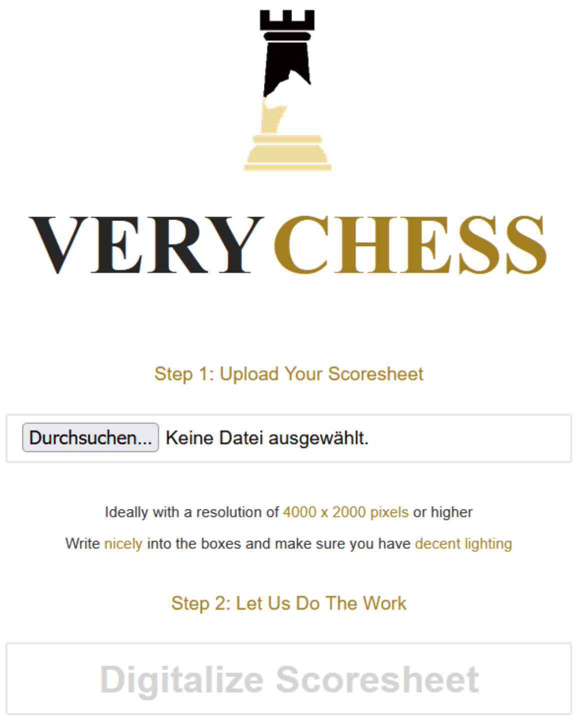


Figure 1: Spinner indicating that the application is working.

On the next screen, the uploaded and aligned picture is shown, correcting the camera angle as best as possible (Figure 3). A progress bar on top show, which step of the OCR the user is currently in. Selecting the blue box which contains the last move of the chess game and confirming triggers the actual moves extraction.

In the moves-editor all the recognized moves are displayed in a table (Figure 4). Using the controls on the left side, the user now has to validate or correct the move suggested by the OCR by comparing it with the original image. Clicking into the moves-table allows for navigation to a specific point in the match.

Once all moves are validated, a form prompts for the matches metadata (Figure 5), and afterwards the finished PGN-file is downloaded.



VERY CHESS

Step 1: Upload Your Scoresheet

Durchsuchen... Keine Datei ausgewählt.

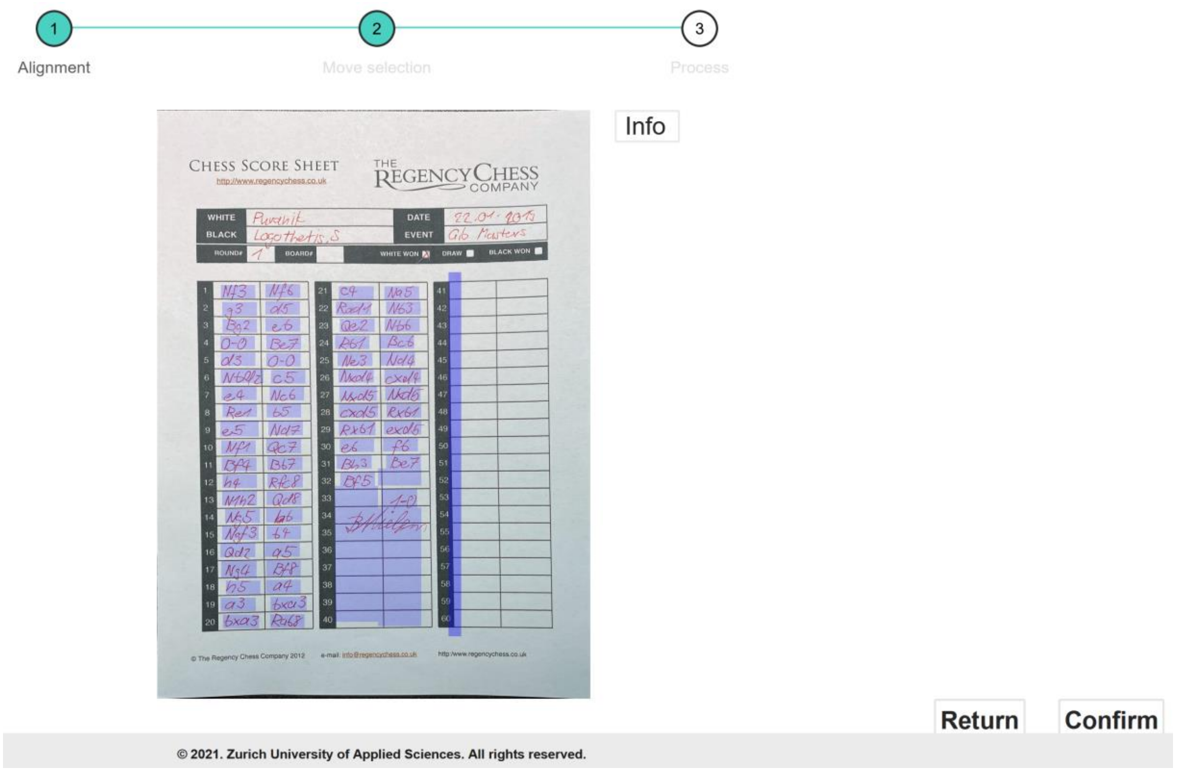
Ideally with a resolution of 4000 x 2000 pixels or higher

Write nicely into the boxes and make sure you have decent lighting

Step 2: Let Us Do The Work

Digitalize Scoresheet

Figure 2: Landing page of the initial "Very Chess" – version



CHESS SCORE SHEET
http://www.regencychess.co.uk

THE REGENCY CHESS COMPANY

WHITE	Purahit	DATE	22.01.2015
BLACK	Logothetis, S	EVENT	Gib Masters

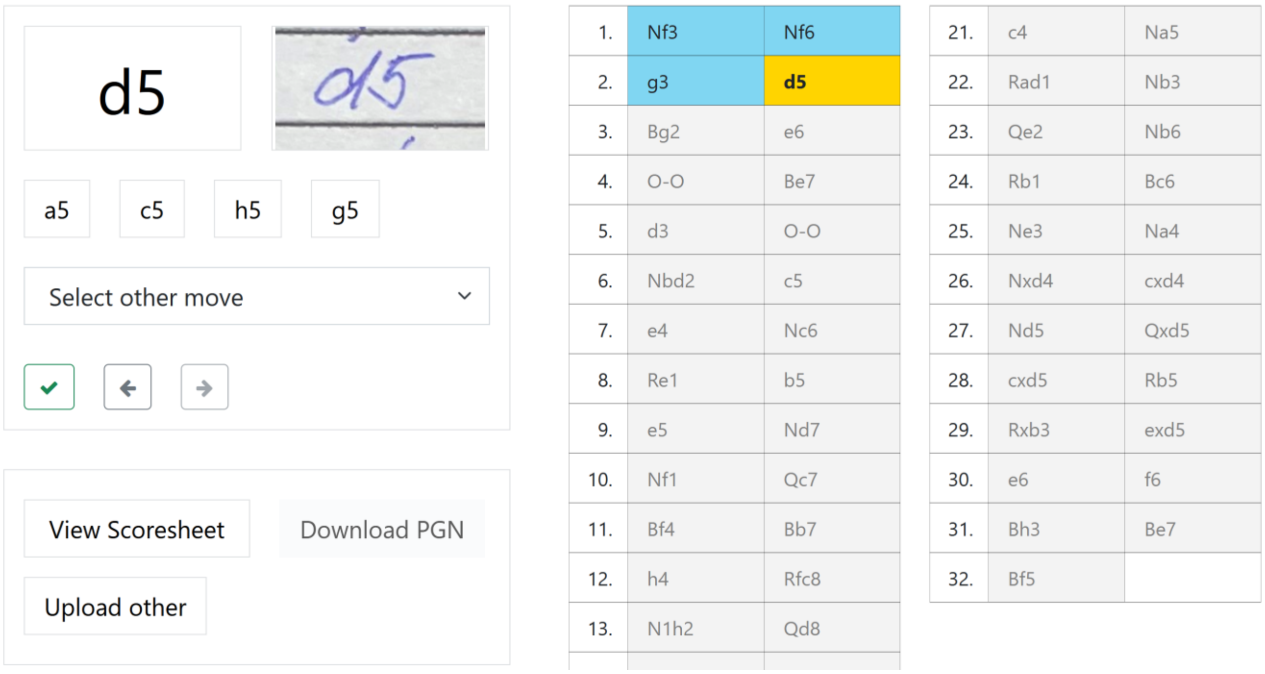
ROUND# 1 BOARD# WHITE WON [✓] DRAW ☐ BLACK WON ☐

1	Nf3	Nf6	21	c4	Na5	41		
2	g3	d5	22	Rad1	Nb3	42		
3	Bg2	e6	23	Qe2	Nb6	43		
4	0-0	Be7	24	Rb1	Bc6	44		
5	d3	0-0	25	Ne3	Nd4	45		
6	Nbd2	c5	26	Nxd4	cxd4	46		
7	e4	Nc6	27	Nxd5	Nxd5	47		
8	Re1	b5	28	cxd5	Rxb1	48		
9	e5	Nd7	29	Rxb1	exd5	49		
10	Nf1	Qc7	30	e6	f6	50		
11	Bf4	Bb7	31	Bh3	Be7	51		
12	h4	Rfc8	32	Bf5		52		
13	N1h2	Qd8	33		1-0	53		
14	Ng5	b6	34	[illegible]		54		
15	Ngf3	b4	35			55		
16	Qd2	a5	36			56		
17	Ng4	Bf8	37			57		
18	h5	a4	38			58		
19	a3	bxa3	39			59		
20	bxa3	Rab8	40			60		

© The Regency Chess Company 2012 e-mail: info@regencychess.co.uk http://www.regencychess.co.uk

© 2021. Zurich University of Applied Sciences. All rights reserved.

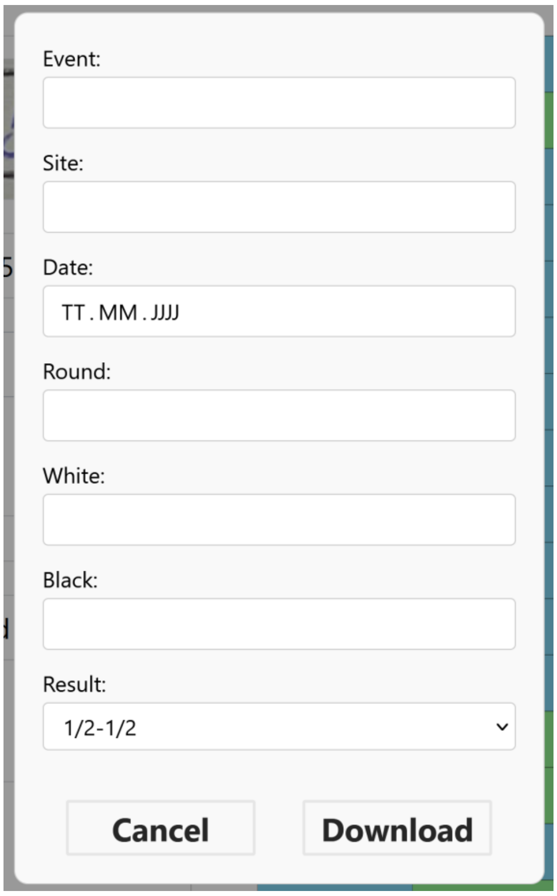
Figure 3: Box confirmation screen of "Very Chess"



1.	Nf3	Nf6
2.	g3	**d5**
3.	Bg2	e6
4.	O-O	Be7
5.	d3	O-O
6.	Nbd2	c5
7.	e4	Nc6
8.	Re1	b5
9.	e5	Nd7
10.	Nf1	Qc7
11.	Bf4	Bb7
12.	h4	Rfc8
13.	N1h2	Qd8

21.	c4	Na5
22.	Rad1	Nb3
23.	Qe2	Nb6
24.	Rb1	Bc6
25.	Ne3	Na4
26.	Nxd4	cxd4
27.	Nd5	Qxd5
28.	cxd5	Rb5
29.	Rxb3	exd5
30.	e6	f6
31.	Bh3	Be7
32.	Bf5	

Figure 4: Original moves-editor page in the prototype



Event:

Site:

Date: TT . MM . JJJJ

Round:

White:

Black:

Result: 1/2-1/2

Figure 5: Form to enter metadata of the game

Limitations

There were multiple limitations in the “Very Chess” application. The website can only digitalize the scorecard of one user at a time. If a second user tries to upload a scorecard at the same time, the process messes up the two scorecards and both users receive mixed data from both scorecards. There is no user management or any option to view past uploaded scorecards, re-download already validated PGN-files or continue an aborted digitalization process.

If one of the externally used APIs does not answer or is not configured correctly, the whole application breaks and no scorecards can be digitalized until the API (or the program code) was fixed, even though the process also works with only some APIs responding. The GUI features various flaws, like sometimes controls half-hidden by bars in certain browsers or just overall difficult ore confusing to use. Overall, the GUI looks rather unprofessional and dated.

As a comparison, the final product of this work, “ChessReader” can be tried at https://chessreader.org (use one of the provided scorecards in the “Additional Files” folder of the hand-in for uploading). Most differences are explained in detail in this document.

1.2 Tasks and Objectives

The official problem description (see 9.1 Official Problem Description) described this work as a follow-up to the preceding ones and stated three possible focal points:

- Analyze which algorithmic steps in the image processing are responsible for errors in the detected moves
- Optimize the corresponding algorithms
- Extend and improve the existing user interface of the web application to make it more intuitive

The main topic of this work was to increase overall quality of the application to a point, that it could be called "production-ready". To achieve this, four areas of improvement have been identified: Stability, security, usability, and performance.

To achieve the given objectives, a list of goals has been defined, based on the results, suggestions, and shortcomings of the previous work by V. Caglayan and B. Nielsen [4].

1.2.1 Stability

Due to the nature of client-server interaction, a web-application must be capable of serving multiple users at once, without the inputs of one interfering with another's process. This has not been the case for the prototype "Very Chess", due to the usage of global variables and saving uploaded images on hard-coded paths, overwriting existing ones in some cases. Also, while a developer can recognize something going wrong by having a look at the logs, any end-user needs to be visibly informed about any errors that interrupt their usage. The following three goals have been defined to counter these issues:

- If three or more users process a scorecard at the same time, no reproducible errors are raised by the simultaneous use (**G-STAB-MULTI**).
- No uploaded scorecard-image is lost due to insufficient handling server-side, while it is being digitalized (**G-STAB-UPPATH**).
- In case of any errors during the process, the user receives an error-message (no never-ending spinners etc.) (**G-STAB-ERRMSG**).

1.2.2 Usability

The official problem description asked for the application to be more intuitive. Since this is a subjective term, fulfilling this task was about gathering information about what potential users might deem "unintuitive", and how to improve it. Consequently, except for (**G-USE-INT80**) and (**G-USE-RESUME**), the goals in this area have been defined after conducting a small field research in a controlled environment, observing how the participants would use the prototype application (see 4 User Feedback).

- At least 80% of a given set of test-users describe the use of the application as “intuitive” (**G-USE-INT80**).
- If the user is logged in, the uploading of a scorecard is saved and can be manually resumed without having to reupload the scorecard-image (**G-USE-RESUME**).
- The user can take a picture directly from a device’s camera to start the process (**G-USE-CAMERA**).
- The interface is responsive to screen sizes, allowing mobile devices to be used for the whole process without major limitations (**G-USE-MOBILE**).
- A virtual chessboard is provided during moves validation to display the current move’s board state (**G-USE-CHESSB**).

1.2.3 Security

One of the requirements that was allowing a user to be able to re-download or resume past uploads. This and the resulting need for a database started prompting security issues. While a thorough security analysis and the design of a comprehensive, secure architecture was out of scope for this work, a simple password-based login process had to be implemented. To comply to laws [5] and major providers’ handling of insecure protocols [6], usage of the HTTPs-protocol and a banner about cookie-usage became mandatory as well.

- Users can create and log into an account to protect their uploads from unauthorized access (**G-SEC-AUTH**).
- The application uses a secure transport protocol (HTTPs) (**G-SEC-HTTPS**).
- A disclaimer informs the user, that cookies are being used and usage-data is saved for potential statistical analysis (**G-SEC-COOKIE**).

1.2.4 Performance

One of the justifications for this application to exist was the gain in time-efficiency when comparing the OCR-digitalization to a completely manual one. This was not always the case in the prototype, mainly due to it sometimes being confident on a wrongly recognized move (false-positive). A goal of improving overall time-efficiency was therefore set.

- The average time to process a given scorecard (including taking and uploading pictures) is reduced significantly in comparison to the prototype (**G-PERF-TIMEFF**).

1.2.5 Discontinued Goals

At the start of this project, additional objectives were taken into consideration, but have been discarded during execution (see 9.1.2 Planning & Timetable). This was due to their implementation demanding too much time to justify the gain, or their fulfillment depending on uncontrollable factors like the responsiveness of external APIs. These include:

- Reading a match's metadata using OCR
- Process a scorecard spanning multiple files
- Never having the user wait for a response for more than two seconds

1.2.6 Renaming the Application

The prototype software was called "Very Chess" by the previous developers. According to the main supervisor, who accompanied all projects on the matter so far, this was an abbreviation for "Verify Chess [Scorecards]", hinting at the process of digitalizing and verifying OCR-recognitions of scorecards. Without this explanation, none of the authors had guessed the meaning of the name though. In direct consequence, a task to rename it appropriately was issued.

For the first approach, the original idea was kept, but simply changing the "y" of "Very Chess" to an "i", to prevent confusion with the word "very". "VeriChess" was unanimously agreed on being an improvement, but the existence of a book with the same name – unrelated to the project – nullified the solution. Instead, the runner-up name was chosen, "ChessReader", to highlight the action of processing (reading) a hand-written chess scorecard.

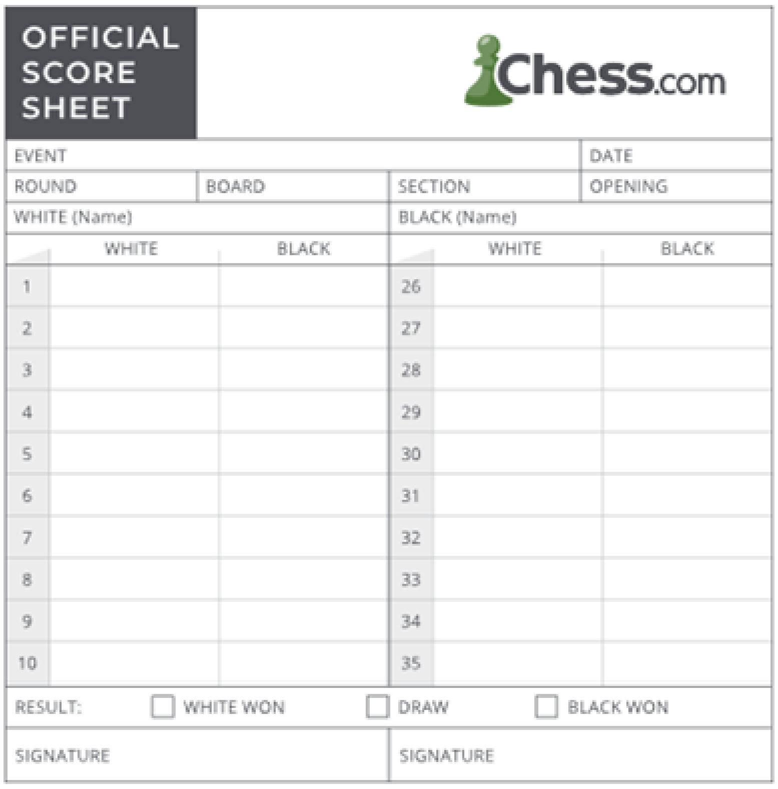
2 Theoretical Background

To understand this work and its predecessors, a base understanding of their underlying principles can be helpful. The following paragraphs aim to explain the related concepts in the chess scorecard process, optical character recognition and artificial intelligence. Knowledge about the rules and details of chess as a game are considered a prerequisite and therefore omitted. A possible source to read up on chess rules would be the International Chess Federation's (FIDE) Laws of Chess handbook [7].

2.1 Chess Scorecards

During official chess games both players are often required to note down every single move executed by both parties on a scorecard (Figure 6 and Figure 7). After the game they sign it, so that it may act as a legal remedy in case of disagreements about the match.

To save time as well as having uniform sheets readable by all participants, the use of standard notations is encouraged or even mandatory. One used or even partially enforced by many organizations, including the FIDE [7], is the Standard Algebraic Notation (SAN).



OFFICIAL SCORE SHEET

Chess.com

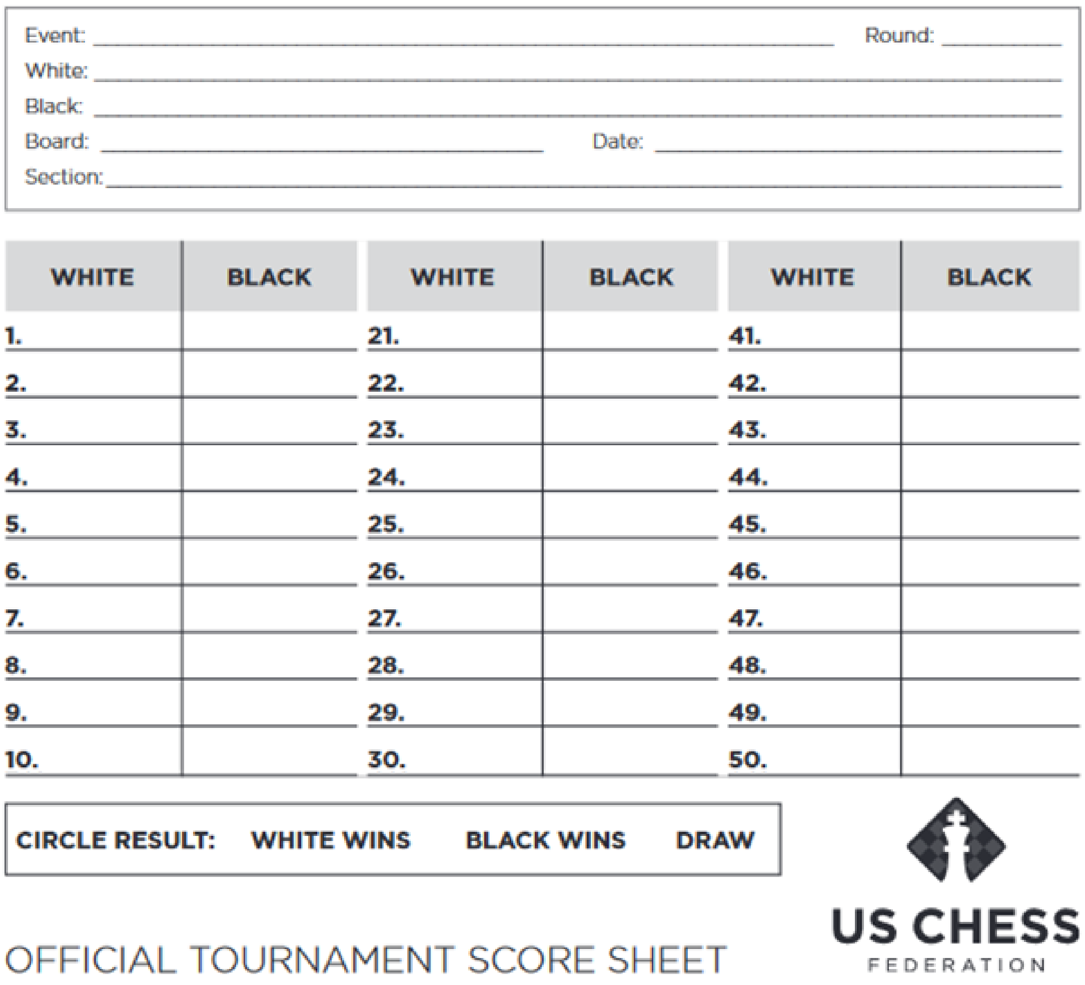
EVENT			DATE
ROUND	BOARD	SECTION	OPENING
WHITE (Name)		BLACK (Name)	

	WHITE	BLACK		WHITE	BLACK
1			26		
2			27		
3			28		
4			29		
5			30		
6			31		
7			32		
8			33		
9			34		
10			35		

RESULT: ☐ WHITE WON ☐ DRAW ☐ BLACK WON

SIGNATURE	SIGNATURE

Figure 6: Part of Chess.com's official scorecard [6] (some table lines omitted)



Event: ______________________ Round: ________

White: ______________________

Black: ______________________

Board: ____________ Date: ____________

Section: ______________________

WHITE	BLACK	WHITE	BLACK	WHITE	BLACK
1.		21.		41.	
2.		22.		42.	
3.		23.		43.	
4.		24.		44.	
5.		25.		45.	
6.		26.		46.	
7.		27.		47.	
8.		28.		48.	
9.		29.		49.	
10.		30.		50.	

CIRCLE RESULT: WHITE WINS BLACK WINS DRAW

US CHESS FEDERATION

OFFICIAL TOURNAMENT SCORE SHEET

Figure 7: Part of the US Chess Federation's scorecard [7] (some table lines omitted)

2.2 Standard Algebraic Notation (SAN)

The SAN is a standardized system of noting movements of chess pieces with the goal of including all information necessary to being able to reconstruct how a match unfolded, while minimizing the number of characters needed to do so. The appendix C of the FIDE Laws of Chess [7] describes it as follows (simplified):

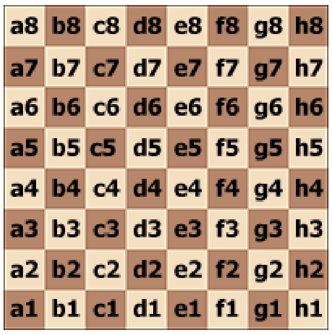
Piece	Icon	Letter
King	♔	K
Queen	♕	Q
Rook	♖	R
Bishop	♗	B
Knight	♘	N
Pawn	♙	

Table 1: List of pieces and corresponding abbreviations.

Icons provided by FIDE [7]

Note: In this description the word “piece” means any chess piece other than a pawn.

1. A move’s note starts with the type of piece that is being moved. All pieces are abbreviated by a single capital letter (Table 1). If a pawn is moved, the letter is omitted.
2. The board is viewed as an 8x8 grid, indicating rows with numbers 1-8 bottom-up, and columns with lowercase letters a-h left-to-right (Figure 8). The abbreviation of the square of arrival usually follows the declared piece.
   Examples: Be5, Nf3, d4.
3. The square of departure can optionally be added first.
   Examples: Bb2e5, Ng1f3, d2d4.
4. If a piece captures another, an x can optionally be added before the square of arrival. If a pawn makes a capture the row it departed from must be indicated.
   Examples: Bxe5, Nxf3, exd4.
5. If two pieces of the same type can move to the same square, indication of the moving piece is necessary. If they are on the same row, the column indicator is added and vice-versa. If they share neither, indicating the column is preferred.
   Examples: Ree5, Ngf3, N1f3, Ngxf3.
6. In the case of a pawn’s promotion, the abbreviation letter of the new piece is appended.
   Examples: d8Q, exf8N, b1B.
7. Castling is noted as either 0-0 (kingside) or 0-0-0 (queenside).
8. Optionally a check-state can be indicated by appending +, checkmate by ++ or #, an offer for a draw by (=).
   Examples: Qd4e3+, 0-0, (=)



a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

Figure 8: How to identify board squares [7]

2.3 Optical Character Recognition (OCR)

The technology to take a picture of or scan a paper sheet - e.g., a chess scorecard – has existed for many years. The resulting file usually just represents a static picture of non-editable text. A lot of effort is necessary for a computer system, to recognize single characters as such and transform the picture into a format processable by text-editors. Flawless success is never guaranteed.

The company ABBYY [8] (see 3.3.1 ABBYY) roughly breaks down the image-processing of their optical character recognition software into four steps:

1. The elements of the document in question are categorized into texts, tables, images, etc.
2. Text-elements are broken down into lines, then words and ultimately single characters.
3. The identified characters are compared to reference images and multiple assumptions are made on what exact character they might be.
4. Based on those hypotheses different variations of breaking lines into words and words into characters are analyzed, and the results compared to actual words to determine the most likely matches. This especially helps on reducing mismatches of similar looking characters.

Intelligent character recognition (ICR) [9] is an advanced form of OCR. To improve the quality and quantity of the recognition of hand-written text, the software uses the picture-inputs to automatically update its database with new handwriting patterns. To achieve this, a self-learning convolutional neural network is often used.

2.4 Convolutional Neural Network (CNN)

Recognizing, linking, and comparing patterns from images is one of the basic approaches a biological brain uses to learn. Artificial neural networks try to reproduce this behavior to develop self-learning algorithms. These aim at optimizing tasks that can hardly – or not at all – be reasonably done with simple, deterministic programming.

Simplified, IBM [10] teaches, that they are comprised of node layers that receive inputs and produce outputs, if certain conditions are met.

As an example, the decision to go surfing might depend on multiple factors, like if the weather is fair and the waves are high. If you're a seasoned surfer, you might not care much about the weather, so a weight is applied to each variable, making some more important for the ultimate decision than others. When the sum of all variables' inputs, multiplied by their weight, exceeds a given threshold, the node is activated and sends its output to the next layer.

So, if the weather is fair (input 1) but its weight is only 2, while the waves are bad (input 0), which is weighted with 5, the resulting sum (1x2 + 0x5 = 2) would not exceed a threshold of 3, therefore the decision would be to not go surfing (output 0). By stacking those layers and using the outputs of one as the input of another layer (Figure 9), increasingly complex decisions can be made.

To improve accuracy, a neural network needs to be trained. This is done by feeding it a vast amount of data and giving it means to determine on how close to the actual solution its decision was. It then changes weights of nodes and thresholds according to mathematical functions to increase accuracy with each iteration.

A convolutional neural network works according to this concept. They specialize in processing pixel data by having at least one layer that works with mathematical convolutions [11]. This makes them a premier choice for recognizing handwritten text, that can then be post-processed into various formats – like PGN – which is widely used for chess software.

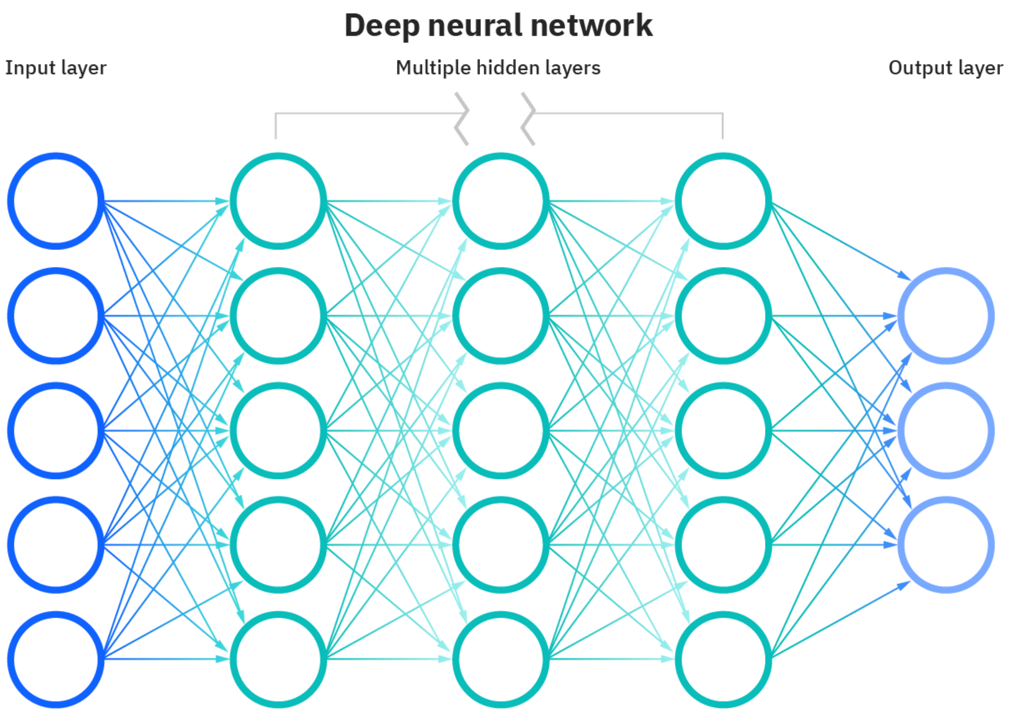
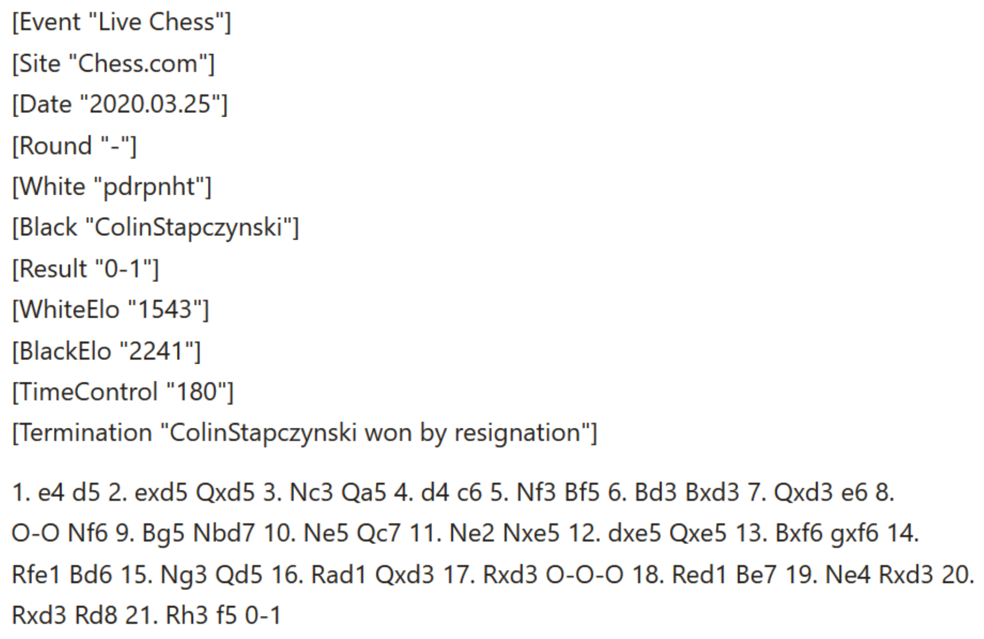


Figure 9: Layering of many nodes to form a deep neural network [10].

2.5 Portable Game Notation (PGN) File Format

According to Chess.com [12], the PGN file format is a text file used to preserve information about how a match of chess played out. It includes all or a part of the executed moves in SAN and metadata like names of the players, location and more (Figure 10).

A PGN-file can be imported into various chess software (like Chess.com) for analysis of the game, commenting, develop and debug programs, and more.



```
[Event "Live Chess"]
[Site "Chess.com"]
[Date "2020.03.25"]
[Round "-"]
[White "pdrpnht"]
[Black "ColinStapczynski"]
[Result "0-1"]
[WhiteElo "1543"]
[BlackElo "2241"]
[TimeControl "180"]
[Termination "ColinStapczynski won by resignation"]

1. e4 d5 2. exd5 Qxd5 3. Nc3 Qa5 4. d4 c6 5. Nf3 Bf5 6. Bd3 Bxd3 7. Qxd3 e6 8.
O-O Nf6 9. Bg5 Nbd7 10. Ne5 Qc7 11. Ne2 Nxe5 12. dxe5 Qxe5 13. Bxf6 gxf6 14.
Rfe1 Bd6 15. Ng3 Qd5 16. Rad1 Qxd3 17. Rxd3 O-O-O 18. Red1 Be7 19. Ne4 Rxd3 20.
Rxd3 Rd8 21. Rh3 f5 0-1
```

Figure 10: Information about a game of chess stored in a PGN-file [12].

3 Methodologies, Tools, Frameworks and APIs

3.1 Work Organization

The whole project was organized using the Agile methodology. The project timeline was split in 7 sprints, each sprint spanning 2 weeks (see 9.1.2 Planning & Timetable). At the start of each sprint, tasks were defined based on the goals issued for the period. Using Gitlab issues (see 3.2.1 Gitlab) the tasks were organized and assigned equally, so that the state of a sprint's progress was always transparent. The project was primarily divided into four parts: User feedback, feature implementation, platform setup and documentation. While both co-authors worked mostly independently on categories according to their preferences and working experience, it was always assured, that implementation intricacies were comprehended by both.

User Feedback

Tasks concerning the planned feedback rounds fell into this category, mainly creating the questionnaire/walkthrough for testing, picking fitting scorecards, as well as conducting and documenting the actual interview with the test users.

Feature Implementation

All coding tasks, independent of if they concerned the front- or backend, were included in this division. This also meant testing the features and assuring a smooth integration into existing code. A separate feature branch was forked for every issue. Merge-requests into development- and production-branches had to be created and were – sometimes more, sometimes less – intensely discussed and reviewed before merging.

Platform Setup

This mainly concerned setting up the tools and server infrastructure used during the project and security tasks. A staging and a production environment were created using Docker on a university-intern server cluster, allowing for Gitlab-integration using Continuous Integration / Continuous Deployment.

Documentation

The final category and the tasks therein were about writing this report and sometimes write short Gitlab wiki entries for clarification about how certain parts worked.

3.2 Tools and Frameworks

Many different tools, frameworks and libraries were used during this and the preceding works, for organization, version control, features, styling, and software architecture.

3.2.1 Gitlab

The official Gitlab instance was chosen as the remote git repository host over a university-intern GitHub cluster. The university's GitHub Enterprise edition did not support integrated continuous integration (CI) / continuous deployment (CD) without a third-party framework. Also, Gitlab offers the option for private repositories without any additional fees, on contrast to GitHub.

Using Gitlab, only a runner installed on a remote accessible server – or one of the publicly shared runners – is necessary to implement CI/CD. It also allows for organizing work easily using issue tracking. A Kanban-style board displays the current state of each issue of the current sprint by using the four stages "Open", "Work in Progress", "Review" and "Done".

If an issue is done, a merge-request can be opened, comparing the changes made with the target branch's files, listing them clearly for reviews. Any developer can "approve" the changes made, signaling to the originator of the request, that they can proceed with the merge. They can also comment and point out flaws of the code on a per-line base, to minimize chances, that the merge will break the existing code; and/or to improve overall code quality. Code reviews also help developers understanding the software they are working with, especially at the beginning of a project. While it can be a significant time investment, it pays off on the long-run by familiarizing developers with all the code that is being worked on, reducing time necessary to implement and debug new (or old) features.

3.2.2 OneDrive

The Zurich University of Applied Sciences' OneDrive hosting was primarily used to store, version and share non-coding artifacts like documents, feedback recordings, references, and other literature between team-members.

3.2.3 Bulma

For designing the responsive GUI to fit all types of screens, especially desktops and mobile phones, the CSS framework Bulma was chosen [13]. It provides ready-to-use frontend components – like navbars, footers, buttons, etc. – that can be combined with a responsive grid layout to create modern looking websites. Compared to more popular choices like Bootstrap or Foundation, Bulma is more lightweight and uses a modular approach with its elements, allowing the designer to integrate only what is used. It also does not feature any JavaScript components, making it a premier choice for pure CSS styling without the risk of having conflicting JavaScript added to the project. Should any dynamic styling be necessary anyways, it is enough to add or remove some of Bulma's numerous CSS-classes using custom JavaScript.

Using frameworks like Bulma makes developing the application easier in many ways. There is no need to design custom components and sticking to the integrated classes provides a coherent look across all parts of the web application. Also, it allows to immediately see the resulting styling of a page by only looking at its HTML code, since all important CSS is applied using classes. Custom CSS is only necessary in rare occasions.

Using Bulma allowed to drop over 1500 lines of custom CSS in the project.

3.2.4 Docker

Docker is a set of Platform as a Service (PaaS) products that uses operating system level virtualization to deliver containers. These can communicate between each other but are otherwise isolated. Each container behaves like a virtual machine, just more practical to use. The major difference is their resource allocation and management. VMs contain fully-fledged operating systems and therefore share their hunger for resources. Containers, on the other hand, share many assets between each other using layers.

Docker sets up a base virtual machine as a foundation for all the containers to share the resources with. Every container includes a stack of layers, each containing a set of instructions, the last one communicating with the running application. Since these layers are shared, e.g., a set of Debian based containers running will share all common data, like the kernel.

3.2.5 Flask

Flask is a lightweight web application framework written in Python [14]. Like many other web frameworks, it features routes that work with models to calculate business functions, usually rendering an HTML or JSON-response as a result and serving them via HTTP(s).

3.2.6 Flake8

One major problem when multiple developers work on the same project is that they all have different coding styles. Therefore, so-called linters like Flake 8 exist [15]. They are "Style Guide Enforcement Tools" and do exactly what the name implies. They enforce certain styling guidelines, like how many empty lines should be in-between methods, or how many spaces should follow a colon. This helps keeping the code clean and readable.

To enforce complying to Flake8's guidelines, Docker can be configured to fail building the web container, if violations of the styling guide are found. This forces the developer to review the code, fixing its styling.

3.2.7 Chessboard.js

Finding a vanilla JavaScript library that offered the chessboard-functionality needed in this project was more difficult than one might think. Chessboard.js nevertheless provides all the tools necessary to display and smoothly animate the virtual chessboard required [16]. What it does not include is parsing moves in SAN. Fortunately, a general chess-library (not about displaying a chessboard) was already included in previous works. It is used to validate the set of possible moves given a certain board state in the moves-editor and can deserialize SAN-noted moves into JavaScript-objects, which then can be transformed into a string processable by the Chessboard.js library.

3.3 External APIs

Multiple external APIs are used for box and character recognition. These were sometimes difficult to work with for various reasons discussed in 7.3.4 API-Reliability.

3.3.1 ABBYY

ABBBYY is a multinational company specializing in artificial intelligence. It offers various products and services. One of which is an API for optical character recognition. ABBYY was the only API used for character recognition in the very first version of the application [2], but was combined with others to yield better results in the last recent one [4].

3.3.2 Google Vision API

The Google Vision API allows to derive insights from images such as detecting emotion, understanding hand-written text and more. “ChessReader” uses it as a mandatory part for both box and character recognition.

3.3.3 Azure Cognitive Services

Developed by Microsoft and belonging to their Azure division, Azure Cognitive Services not only allows to work with images, but also offers a multitude of different services, that use artificial intelligence and machine learning to work with speech, language, vision, and decision making. In this application its computer vision service is used for box and character recognition.

3.3.4 Amazon Rekognition

Belonging to the Amazon AWS division, Amazon Rekognition is a cloud-based service for image recognition processing texts to objects and facial emotions. In this case it’s used for both box and character recognition.

4 User Feedback

The product of this work is a revised, improved version of an already existing prototype. Preceding projects had worked hard on optimizing the OCR algorithm for the task at hand. Therefore, this one's focus lied on developing a production-ready tool. The academic part relied on receiving valuable user feedback on the prototype, to determine what changes and improvements had to be made in the GUI, so that usability is always guaranteed.

4.1 Goals

The overall goals of the project (see 1.2 Tasks and Objectives) specified, that the resulting application should be more intuitive when compared to the prototype. Also securing that the use of it results in a gain in time-efficiency, when compared to the manual digitalization of a scorecard, was compulsory. Consequently, the user feedback served to achieve the following goals:

- Determine, what parts and processes of the application are deemed to be unintuitive and/or time-consuming.
- How to improve the GUI to make it more intuitive.
- How to improve the GUI and overall process to make it less time-consuming.
- Compare the time-efficiency of the overhaul with the original version.

Ultimately it had to act as a base to formulate the usability goals for the project (see 1.2.2 Usability), as well as confirming that the performance goals (see 1.2.4 Performance) were fulfilled.

4.2 Procedure

To conduct the feedback a questionnaire was written, including instructions for the interviewer on how to guide the tester through the process. The template for the questionnaire can be found in the “Additional Files” folder of the hand-in. Since the research was based on participation observation, one of the project's authors had to always be present, resulting in a severe time commitment for each test person.

The feedback procedure was divided into two parts, to adequately deal with the usability objectives, as well as the performance ones. Three specific, previously filled-in chess scorecards were used over both parts. They can be viewed in the “Additional Files” folder as well.

4.2.1 Usability-Testing

The usability part was set up like field research in a controlled environment. The test person was presented the first scorecard (#1) and tasked with digitalizing it, using the application, without any further introduction to it, to ensure uninfluenced results about its intuitiveness. All the while, the observer would take notes on how the software was used and ask questions about the participant's reasoning and thought process on navigating it.

After gaining some experience, the test user was tasked with processing a less cleanly written scorecard, to see how they would cope with false-positives and the tools the tested version of the application offered (or the lack thereof) to find and deal with them. If no false-positives appeared during the preceding scorecards, the user was given a short explanation of how and why they happened.

During the second feedback round, the usability test was less structured, turning into a feature evaluation to answer the question of the application's intuitiveness.

4.2.2 Performance-Testing

To test the improvement of time-efficiency when using "ChessReader", testers were asked to process two scorecards (#2 and #3), focusing on doing it fast but also without any errors, after familiarizing themselves with the GUI in the first usability test.

The cards were of increasing difficulty, aiming at raising false-positives from the OCR, as well as having the test users think about what the correct move in the given situation was, by adding poor handwriting and even mistakes on the written sheet. This was meant to simulate a situation, where the games played had taken place sometime in the past, where the player would not immediately remember the moves done. This part of the feedback was conducted again with an improved version of the application, after the inputs of the first were accounted for. The second run served as a comparison to the initial performance of the software, supporting the judgment of the performance-goals' fulfillment.

4.2.3 List of Test-Users

To create a continuous feedback-loop during the implementation of the update, a small group of people agreed to test the application during multiple iterations of the development cycle.

User	Name	Involved in
TU01	Nico Ambrosini	Development process, first feedback, second feedback
TU02	Michael Blum	First feedback, second feedback
TU03	Gundula Heinatz	First feedback, second feedback
TU04	Gian Hellinger	Development process, first feedback, second feedback

Table 2: List of test-users

4.3 Results of First Feedback

The first feedback round was concluded on May 18, 2022, after about 70% of the available time for the thesis had passed. The version tested was not the original prototype, but an interim version, that already featured multiuser-support and Bulma’s color scheme. Layout and styling of the content was still very similar to the original though, except for the colors. Recordings of the feedbacks, as well as the notes taken, can be found in the “Additional Files”-folder of the hand-in. Results from the performance-test and the feature evaluation (second usability feedback) are presented in 6 Results. Results for the first usability feedback heavily influenced the development of “ChessReader”’s finalizing features, which is why they are presented separately from the end-results.

The first usability feedback round yielded three types of valuable information: Opinions on existing features, suggestions for changes or additions, and bugs discovered.

4.3.1 Opinions

Every user feedback yielded a large amount of valuable information about potential users’ ideas about what is happening on screen while digitalizing a scorecard. There were various points that left the testers confused or lost:

- With no scanner in close reach, a mobile phone’s camera is the most intuitive way to take a picture of a physical scorecard, but how to transfer the image to a PC afterwards – since the interface is only optimized for desktop screens – is up to the user’s imagination. (**FB-IMO-FILE**)
- Without any information, what needs to be done during box confirmation is very vague and intuition leads to simply pressing “Confirm”, because everything seems alright. (**FB-IMO-BOXINF**)
- The moves-editor provides a lot of information and controls, but none of them are explained. Also, there is no explanation of what the goal of this step ultimately is. (**FB-IMO-MOVINF**)

Some issues concerned the controls provided by the application:

- The “checkmark”- and arrow-controls are never used, since they are too far away from where the eyes attention is (on the big suggestion-button, the provided OCR-picture and/or the moves-table). (**FB-IMO-ARROWS**)
- The 4 additional suggestion buttons are less attention-grabbing than the dropdown-list with the remaining moves, since they are only outlined, while the list has a fully filled white background. (**FB-IMO-4SUGB**)

- The moves-table often overflows the screen on the bottom, therefore a lot of scrolling needs to happen, when validating moves that are in the lower rows. (**FB-IMO-SCROLL**)
- Pressing the "Download PGN"-buttons prompts for entering metadata instead of downloading the PGN-file. (**FB-IMO-PGNDL**) While not a big issue, this was deemed counter-intuitive.
- The dropdown-list with the remaining moves is difficult and slow to use, due to the lack of ordering and filtering. (**FB-IMO-DDL**)

For the last point, two observations were made by the interviewers, that were confirmed by the test users during questioning:

- Proficient chess-players tend to intuitively validate a move by how much sense it makes in the game's current state and do not actively compare the suggested move with the provided image of the scan. (**FB-IMO-VALMOV**)
- Validating and correcting moves is by far the most time-intensive part of the process, especially when backtracking needs to be done due to false-positives happening. (**FB-IMO-FALPOS**)

4.3.2 Suggestions

Many suggestions were made during the feedback, some addressing the digitalization process as a whole:

- Add a chessboard displaying the current moves state. (**FB-SUG-CHESSB**)
- Animate every move on the chessboard, instead of skipping confident recognitions. (**FB-SUG-ANIME**)
- Make it possible to drag-and-drop pieces on the chessboard to correct/validate the current move. (**FB-SUG-DRAG**)
- Use a device's connected camera to take scorecard-pictures, to avoid having to transfer files. (**FB-SUG-CAMERA**)
- Provide more information about what tasks are at hand and what specific controls do, for example using one-time modals. (**FB-SUG-MANUAL**)
- Design an optimized version of the application layout for mobile devices, to be able to digitalize without a PC present, and to avoid having to transfer files. (**FB-SUG-MOBILE**)
- Automatically display the metadata-form after the last move was validated. (**FB-SUG-META**)

Others were directed at specific controls of the application:

- Use a text-input with autocompletion instead of a dropdown-list for the remaining valid moves. (**FB-SUG-DDLFTI**)

- Add a meaningful order to the suggested moves in the dropdown-list (or in the autocompletion respectively). (**FB-SUG-DDLORD**)
- Replace the moves-table with a scrollable single-column list, that always has the current move in the center, showing a fixed number of rows before and after. (**FB-SUG-MOVLST**)
- Use weaker colors in the moves-table to not draw that much attention. (**FB-SUG-PASTEL**)
- Remove unused, redundant controls entirely. (**FB-SUG-RMCON**)
- Reformat the 4 additional suggestion buttons to be more prominent than the dropdown-list. Also move them closer to where the eye looks for comparing suggestions. (**FB-SUG-VIP4SB**)

The last group of suggestions comprised feature requests. Due to the lack of time remaining, these were – except for the animated chessboard (**FB-SUG-CHESSB**) – considered out of scope and are described in 7.5.6 Additional Ideas.

4.3.3 Bugs

Unfortunately, the version tested featured a series of bugs, most of which did not have a serious impact on achieving the feedback goals:

- The match result in the resulting PGN-file was always “1-0”, independent of the user’s input.
- The colors red and blue were inverted on the uploaded image.
- Boxes containing information on what to do were not showing
- Partial images of the scan were rather small, due to columns being low (no bug but an oversight in the design)

These were all addressed and fixed properly during implementations following the feedback. Two bugs however had to be worked around during the interviews:

- Due to the low row-height, the OCR sometimes thought to be recognizing moves in-between moves, resulting in the moves-table having too many rows, with no option on removing the additional ones.
- The uploaded image was sometimes rotated 90 degrees, if not all edges of the sheet were visible on the picture taken from a scorecard.

The first was ignored since it had no impact on the testing process. The resulting PGN would be invalid, but the whole digitalization could still be executed properly.

If the second arose, the picture would be retaken, making sure, that all edges would be visible, and uploading again.

4.3.4 Conclusions

The seemingly most important requests were formulated into committed overall objectives, while minor ones – or if how to implement them was not decided right away – were considered parts of the application having to be intuitive (**G-USE-INT80**). Correlating suggestions to opinions, the first three goals set for the first user feedback were concluded, and overall usability-goals added as follows:

Part / Opinion	How to improve	Conclusion
Transfer file to PC (**FB-IMO-FILE**)	Make process completable using a single desktop or mobile device (**FB-SUG-CAMERA**) (**FB-SUG-MOBILE**)	More viable for mobile than for desktop devices. Added goals (**G-USE-CAMERA**) and (**G-USE-MOBILE**). Further information: 5.1.4 Usage of Connected Cameras 5.1.5 Responsive Design
Missing Information (**FB-IMO-BOXINF**) (**FB-IMO-MOVINF**)	Provide more information using one-time modals (**FB-SUG-MANUAL**)	Modals showing a manual until acknowledged (uses cookies). Covered by (**G-USE-INT80**). Further information: 5.1.7 Manuals
Information overload in moves-editor (**FB-IMO-ARROWS**) (**FB-IMO-4SUGB**) (**FB-IMO-SCROLL**) (**FB-IMO-MOVINF**)	Remove unused controls to unclutter the interface. Reduce amount of strongly colored parts. Reduce moves-table in size and make it scrollable. Reformat suggestion buttons. (**FB-SUG-MOVLST**) (**FB-SUG-PASTEL**) (**FB-SUG-RMCON**) (**FB-SUG-VIP4SB**)	Removed “checkmark”- and arrow-buttons, recolored various controls with lighter colors. Covered by (**G-USE-INT80**). Further information: 5.1.1 Interface Design 5.1.2 Moves-Editor Layout
Wrong action triggered (**FB-IMO-PGNDL**)	Make the download-button actually trigger downloading the file. Show metadata-form	Covered by (**G-USE-INT80**). Further information: 5.1.2 Moves-Editor Layout.

	automatically after validation. (**FB-SUG-META**)	
Dropdown-list slow (**FB-IMO-DDL**)	Replace the dropdown-list with a text-input with autocomple-tion. (**FB-SUG-DDLFTI**) (**FB-SUG-DDLORD**)	Covered by (**G-USE-INT80**). Further information: 5.1.2 Moves-Editor Layout.
Most time is spent on fixing false-posi-tives (**FB-IMO-FALPOS**)	Center the validation process around an animated chess-board, where every move is shown instead of skipping by confidence, to prevent false-positives. (**FB-SUG-CHESSB**) (**FB-SUG-ANIME**) (**FB-SUG-DRAG**)	Using drag-and-drop on the chess pieces to validate or correct moves was not developed because of time restrictions. Added goal (**G-USE-CHESSB**). Further information: 5.1.2 Moves-Editor Layout 7.5.1 Chessboard Drag-and-Drop

Table 3: Results of first feedback round and how they were addressed

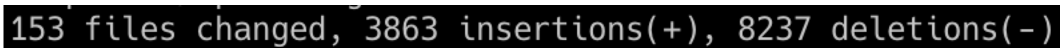
The observation made about how the main target audience uses the moves-editor (**FB-IMO-VALMOV**) unveiled a crucial flaw in the feedback process, that is being discussed in 7.1 Interpre-tation and 7.3.1 Using A Fixed Set of Scorecards for User Feedback.

The feature evaluation feedback is presented in 6.1 Acceptance Test Protocol, the performance evaluation in 6.3 Performance.

5 Implementation

This paragraph explains the technical implementation and shows examples of all the changes that have been made to the original version of “Very Chess”, during its transformation to the current iteration of “ChessReader”. According to this work’s focus, this mostly contains changes to the frontend and the database-implementation. For information about basic features, original goals and details of the underlying OCR-algorithm (backend), please refer to 1.1.2 Base Application “Very Chess” and the previous works of V. Caglayan and B. Nielsen [4], B. Horváth and C. Dreher [2] and A. Abduli [3].

All changes to the code can be viewed in detail (every single line) in the code versioning tool used (see 3.2.1 Gitlab) by comparing the “main”-branch to the “original version”-branch (Project access necessary). It includes over 12’000 changed lines when comparing the final product to the original, without interim-versions.



```
153 files changed, 3863 insertions(+), 8237 deletions(-)
```

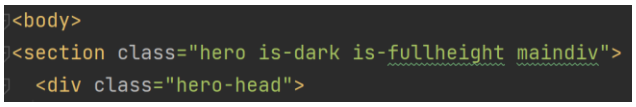
Figure 11: Number of lines changed when comparing "ChessReader" to "Very Chess", according to git.

5.1 Frontend

The frontend of a web application includes the (visible) interface the user is interacting with, including layout, color design, text formatting, etc., and the collection, packaging and pre-processing of data sent to the server-side backend. It runs in the user’s browser and therefore uses the client’s computation resources. Changes in the frontend are usually the most prominent, since it is what a user sees and uses to navigate when using the application.

5.1.1 Interface Design

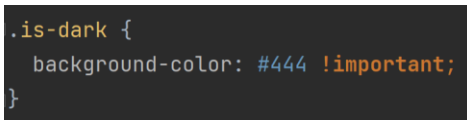
“ChessReader” features a new color scheme in comparison to “Very Chess”. Modern trends go towards using dark colors for most areas of the screen, because they are easier on the eyes, especially with low or bad lighting. Bulma offers a premade color scheme [17], including a dark mode by simply adding the “is-dark” class to the main content div (Figure 12). Since there were no requirements for specific colors and composing them appropriately is a difficult task usually done by trained designers, making the task out of scope for this work, it was decided to stick with the basic palette, using teal as the primary color.



```
<body>
<section class="hero is-dark is-fullheight maindiv">
  <div class="hero-head">
```

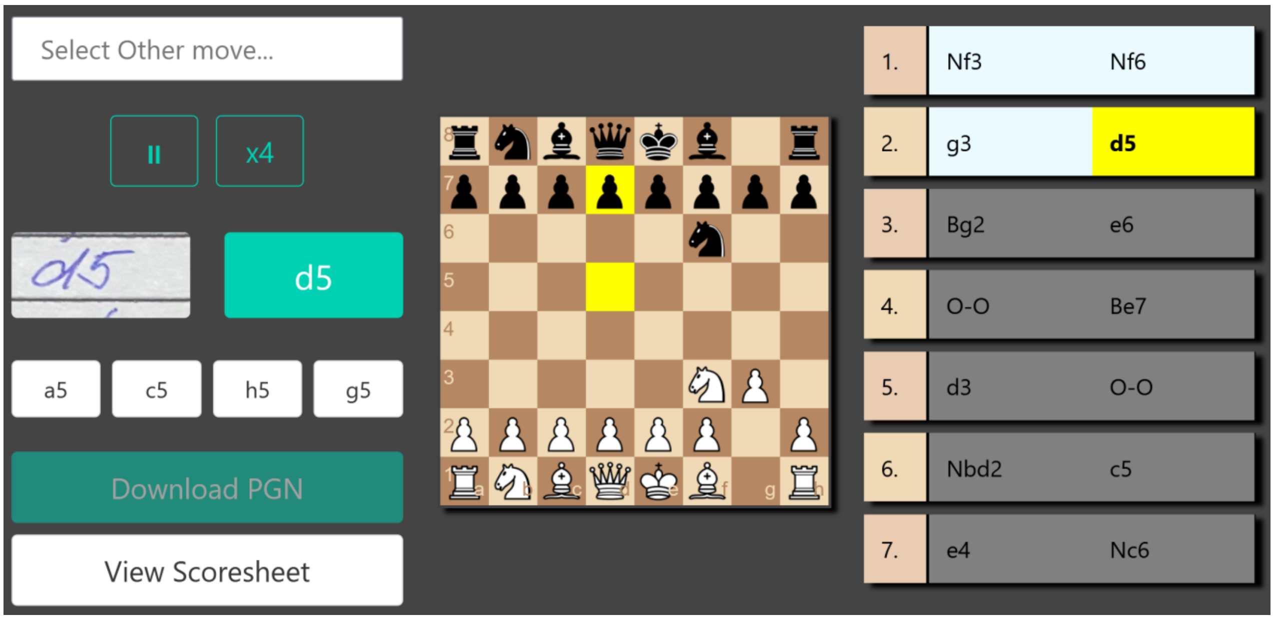
Figure 12: Using Bulma's dark mode by adding the "is-dark" class

The default background color had to be lightened up a little though, to provide enough contrast to the black in the application's logo. Like with most modern CSS-frameworks, the colors used could be adjusted by simply changing SASS-variables [18]. But since only vanilla CSS has been used in the software so far, and to not increase the overall complexity further, including SASS was discarded and the color overwritten by a simple line of custom CSS (Figure 13).



```
.is-dark {
  background-color: #444 !important;
}
```

Figure 13: Overriding the background-color using CSS



1.	Nf3	Nf6
2.	g3	**d5**
3.	Bg2	e6
4.	O-O	Be7
5.	d3	O-O
6.	Nbd2	c5
7.	e4	Nc6

Figure 14: The new application design, featuring a dark background and using teal as primary color

Buttons and other Controls

In addition to the new color scheme, buttons and other controls have been formatted according to their hierarchical importance in the default flow of the software using modern standards. Important controls, which the designer wants the user to primarily use, are highlighted to grab their attention. As depicted in Figure 15, the only difference in the original button-controls were varying back-ground-colors, while being hovered by the mouse cursor, if at all.

The “check mark”-button with the green border is the one that seems to be the most important due to the different coloring, but since it is a lot smaller than the button on top, it fails to attract any attention. The big “d5”-button in fact triggers the same action, therefore has the same hierarchic value, but is formatted differently. Since they serve the same function, removing one of them to de-clutter the interface, is also a valid option.

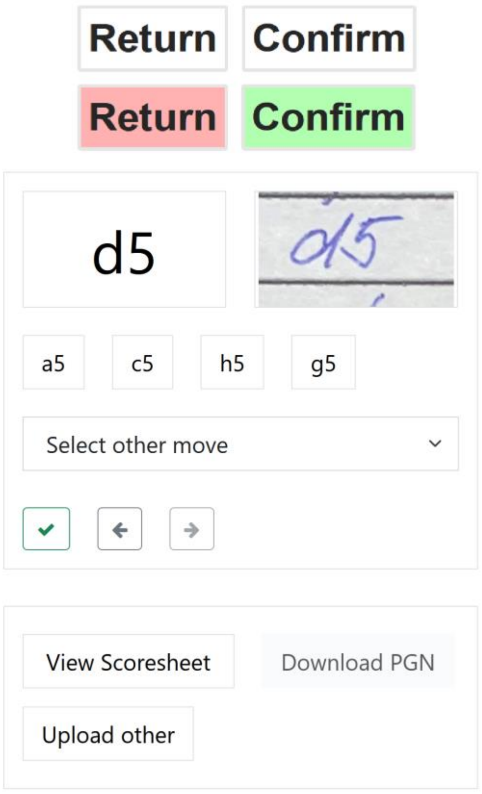


Figure 15: Non-hierarchic button controls in the original

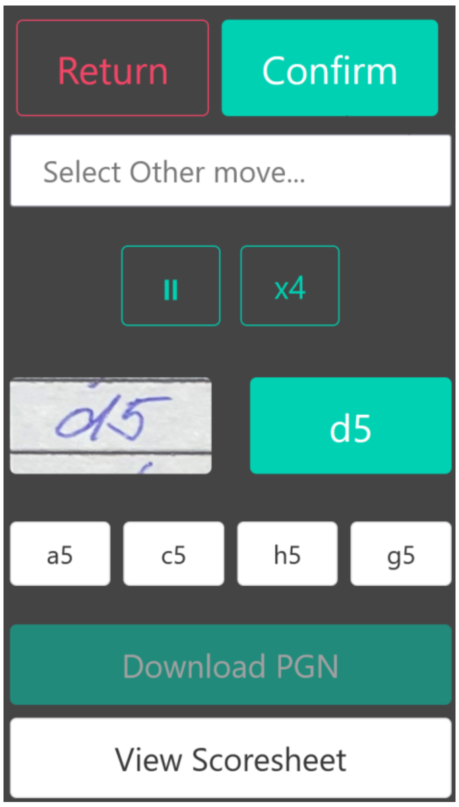
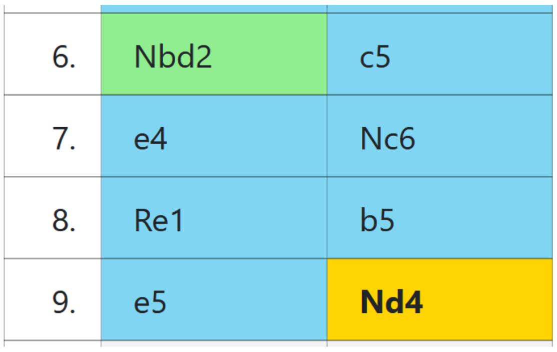


Figure 16: Reworked controls using primary colors

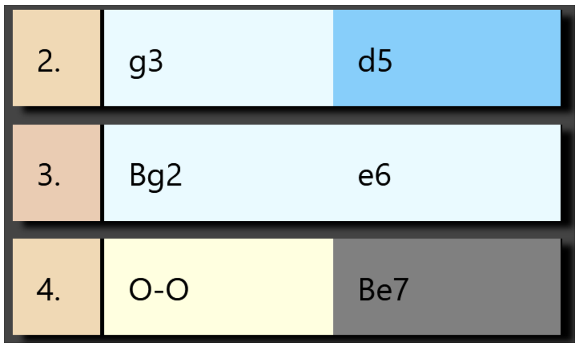
The reworked controls (Figure 16) highlight the primary positive actions (confirm box selection, validate suggested move, download finished file) by filling their background with the app’s primary color, to intentionally direct the user’s sight onto them. Secondary positive actions (correct move with suggestions or autocompletion, view original scorecard image) have a more neutral background to not divert attention from primary actions, while features not crucial to the main process (pause/speed up animation) are only outlined. Potentially harmful actions (returning to the landing page) are outlined in red to indicate a possible data loss.

The moves-editor’s table also used a lot of strong signal colors (Figure 17), which, in combination with the automatic animation, sometimes distracted users. Figure 18 shows the revised version, which uses lighter, less noticeable shades, light blue for skipped moves, blue for corrected moves and light yellow for the current move. If manual input is needed for validation, the light-yellow turns into a full, brighter yellow to signal the need for action (Figure 19).



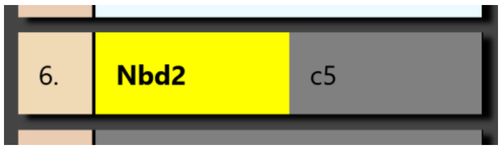
6.	Nbd2	c5
7.	e4	Nc6
8.	Re1	b5
9.	e5	**Nd4**

Figure 17: Moves table using strong signal colors blue, green and yellow



2.	g3	d5
3.	Bg2	e6
4.	O-O	Be7

Figure 18: Revised color scheme of the moves list



6.	**Nbd2**	c5

Figure 19: Bright yellow signaling the current move needs to be manually validated

5.1.2 Moves-Editor Layout

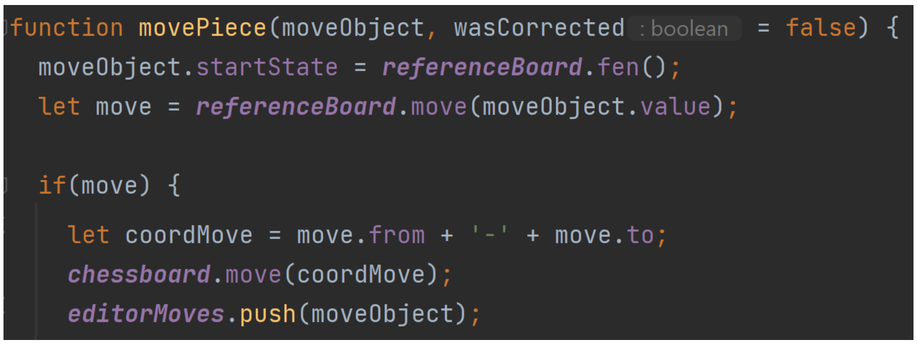
In addition to a revised color palette, the moves-editor experienced a multitude of functional updates.

Chessboard

The central addition is an animated chessboard. Instead of skipping all recognized move until one is found, where the OCR was not confident about its result, every confident move is now being played through automatically in sequence. The idea behind is, to have the user verify – or at least throw a glance at – every move, to reduce the chance of false positives happening.

During the first feedback round, false positives have been identified as the most impactful source of large delays in the digitalization process (see 4.3 Results of First Feedback). The time lost by animating every move one-by-one is easily made up if it allows to prevent even a single false positive. Also, it helps the player to relive the match and therefore verifying how it played out holistically.

The code uses the chessboard.js library to create and animate the board (see 3.2.7 Chessboard.js). Unfortunately, it does not feature a parser that accepts a move in SAN, therefore the Chess.js library is used – like in the moves-editor as well – to translate and validate the moves, before being fed to the board in a processable format (Figure 20). It also saves the current state of the board before execution, so that it can be reloaded when the moves-list is used to jump back to this state.



```
function movePiece(moveObject, wasCorrected : boolean = false) {
  moveObject.startState = referenceBoard.fen();
  let move = referenceBoard.move(moveObject.value);

  if(move) {
    let coordMove = move.from + '-' + move.to;
    chessboard.move(coordMove);
    editorMoves.push(moveObject);
```

Figure 20: The moves-editor using Chess.js (referenceBoard) to translate and validate a move, before executing it on the animated board (chessboard)

The next move is highlighted by coloring the partaking squares grey (see Figure 22) and a delay of two seconds allows for a quick verification, before the animation happens and the game advances. If the editor stops to wait for manual validation, the proposed move is displayed in yellow instead, the same color as in the moves-list (see Figure 21).

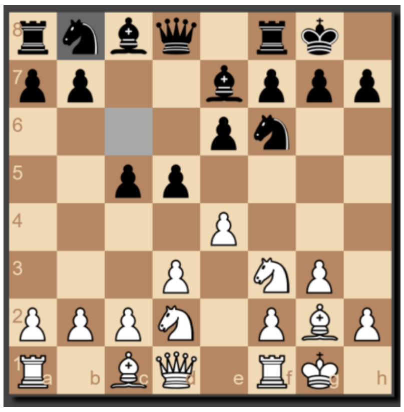


Figure 22: Chessboard highlighting next move with grey squares

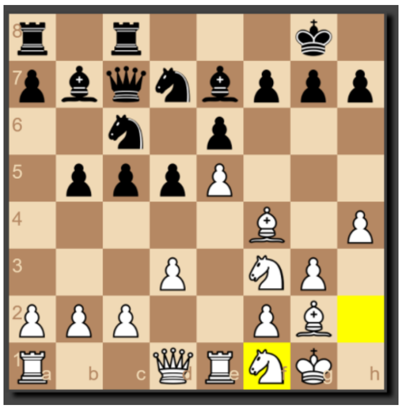


Figure 21: Chessboard highlighting move to be validated in yellow squares

Editor Controls

Apart from the color changes, the original controls (Figure 23) featured a “check mark”- and two arrow-buttons, which have been removed. User feedback showed (see 4.3 Results of First Feedback), that they were barely ever used. The dropdown-list with additional suggestions was replaced by a more intuitive text-input, that features auto-completion for valid moves (Figure 24).

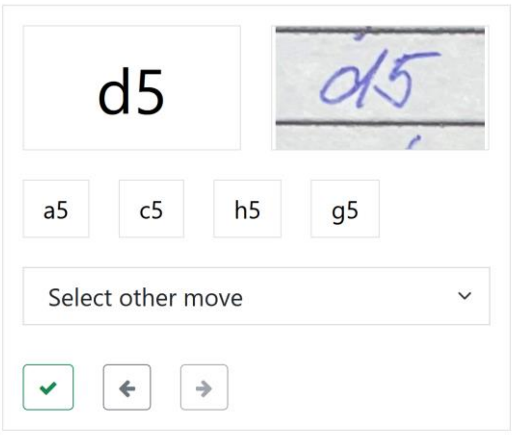


Figure 23: Original moves-editor controls

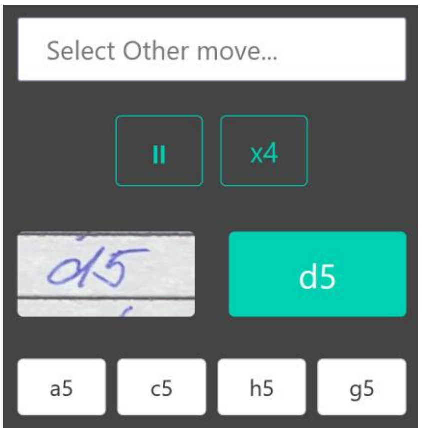


Figure 24: Revised set of controls missing "check mark"- and arrow-buttons, but featuring animation-controls

While moves are being executed on the chessboard, most controls are unavailable and therefore greyed out. The exception is the two small, outlined buttons, which control the animation of the moves. Using the pause-button (or the space bar) stops the automatic advance and enables manual controls, starting with the current move – hitting it again (now having changed to a play-button) resumes the playthrough. Clicking on the “x4”-button (or the up-arrow-key) reduces the delay between moves to a fourth of the original and changes its text to “x1”. Another click (or pressing the down-arrow-key) adjusts it back to the original value.

Validation of the suggested move is still done by either clicking on the primary button (the big, teal one) or hitting the “Return” key on the keyboard. Navigating to the next or previous moves can also be done using the left and right -arrow keys on the keyboard. This is very intuitive behavior for chess players, due to the similarity to other chess platforms. Navigating forward also validates the suggestion if it is the move currently being checked. Using this method also stops the auto-advance, passing control over the validation speed to the user.

The 4 additional suggestions button still show the 4 most likely – according to the OCR – alternate moves. A click on them corrects the current move and re-evaluates the proposed moves for the rest of the match. All other possible moves that are valid on the current game state are available through the top text-input.

Text-Input with Autocompletion

Since not all moves are directly displayed on the page, a dropdown-list is necessary to show more than 5 suggestions. Therefore, a standard HTML select was implemented in the prototype. Since they were displayed in seemingly no particular order, it was difficult to find the move being searched for. Also, if there were still a lot of moves possible in the current board state, the vast list of suggestions made it even harder.

To solve all these issues in one fell swoop, the dropdown-list was replaced by a text-input with autocompletion. It acts like an HTML select tag, displaying all possible moves in a list when clicking into it (Figure 26), but automatically reducing the suggestions matching what is being typed into the input (Figure 25).

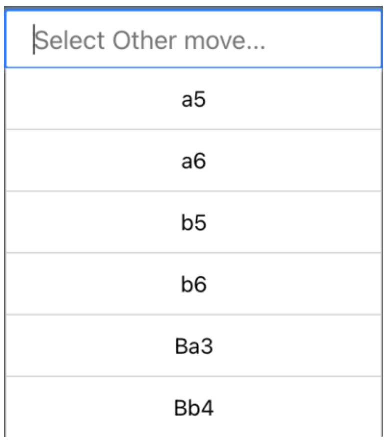


Figure 26: Selection box without any user input.

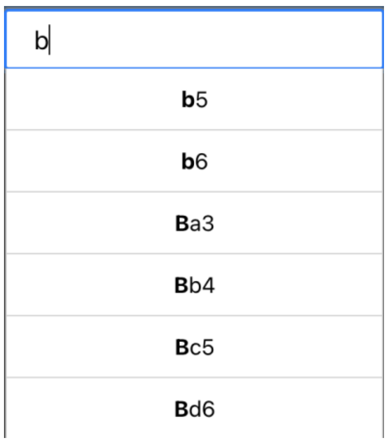
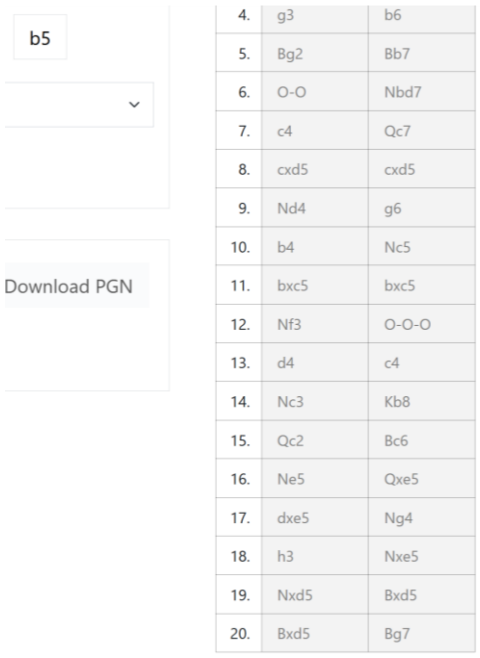


Figure 25: Selection box with a custom user search. The search is not case-sensitive, this allows the user to search faster.

Moves List

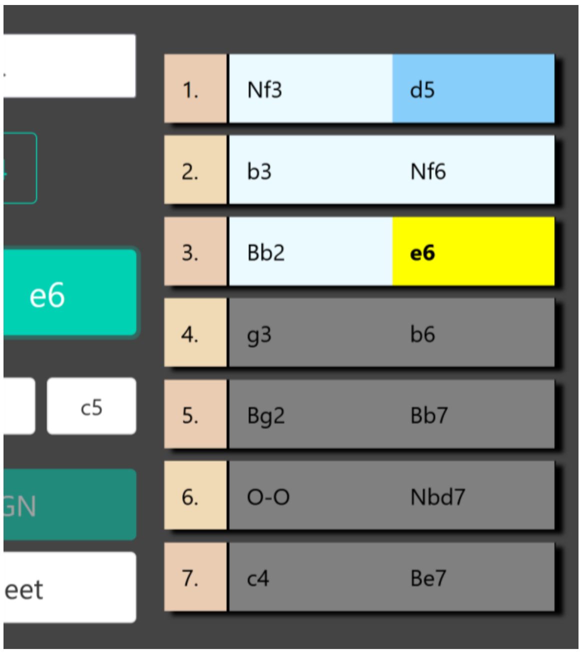
The original moves-table copied the layout of the actual table on the scanned scorecard. Therefore, if the scorecards had 40 moves distributed on two columns of 30 moves each, the table shown in the application would have similarly sized columns, making scrolling necessary, if a column did not fit the screen (Figure 27). To solve this problem, and make room for the chessboard in the GUI, all moves are now listed in a single, scrollable column (Figure 28). Instead of scrolling the whole website, possibly having the editor controls off-screen afterwards, the list now automatically jumps to the current move with a smooth animation. In the default desktop layout, the current move, the previous 4 and the subsequent 2 rows are shown, so that its height does not exceed that of other parts of the GUI.

The moves-list is still used to navigate to a given game state by clicking on it. This also stops auto play and updates the pieces on the board to the line-up they were in before the move took place.



4.	g3	b6
5.	Bg2	Bb7
6.	O-O	Nbd7
7.	c4	Qc7
8.	cxd5	cxd5
9.	Nd4	g6
10.	b4	Nc5
11.	bxc5	bxc5
12.	Nf3	O-O-O
13.	d4	c4
14.	Nc3	Kb8
15.	Qc2	Bc6
16.	Ne5	Qxe5
17.	dxe5	Ng4
18.	h3	Nxe5
19.	Nxd5	Bxd5
20.	Bxd5	Bg7

Figure 27: Original moves-table exceeding the rest of the GUI (and possibly the screen) in height

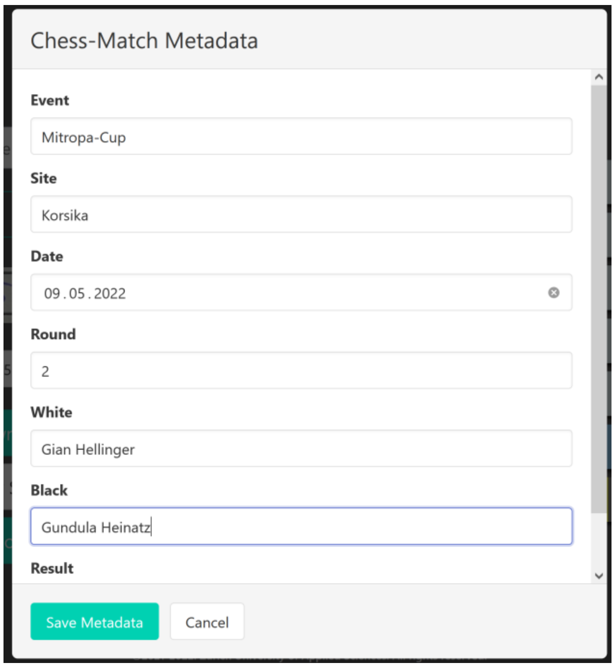


1.	Nf3	d5
2.	b3	Nf6
3.	Bb2	**e6**
4.	g3	b6
5.	Bg2	Bb7
6.	O-O	Nbd7
7.	c4	Be7

Figure 28: New scrollable moves-list, having (roughly) the same height as the controls

Metadata Form

After the whole game has been validated, the user is automatically prompted to enter metadata about the match by a modal overlay. Apart from some layout- and formatting-changes, this is still the same as before (Figure 29). The data collected however is now sent to the backend and stored in the newly integrated database (see 5.2.1 Database).


Chess-Match Metadata

Event
Mitropa-Cup

Site
Korsika

Date
09.05.2022

Round
2

White
Gian Hellinger

Black
Gundula Heinatz

Result

Figure 29: Metadata-form that submits data to the backend when saved

Meta Controls

Additionally, to the inputs that directly control the editor, there are three buttons that trigger meta-actions (Figure 30). The first allows to view the uploaded image in a separate tab or window, the second delivers the match as a PGN-file after completion and the last returns the user to the home screen to digitalize another scorecard. Functionally these are basically the same as previously, but the last button was deemed to be potentially harmful, since it cancels the current edit, if not all moves have been validated yet. Therefore, it is now hidden until validation of the current scorecard has been finished (Figure 31 and Figure 32). The index page is reachable through the top navigation bar, aborting the ongoing process.

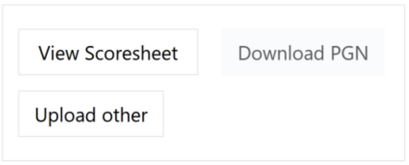


Figure 30: Original meta-controls to view the uploaded image, download the result or restart the process with another scorecard

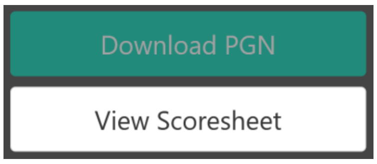


Figure 31: Newly formatted meta-controls with hidden "Upload Other" - button

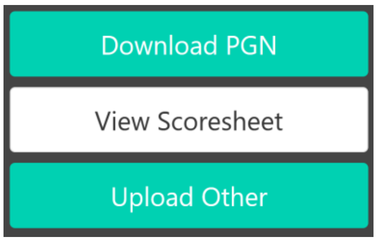


Figure 32: Meta-controls after the match has been validated, showing "Upload Other" and enabling the PGN-download

5.1.3 Top Navigation Bar

To ease access to important parts and functions of the application, a navigation bar has been added to the top of the screen.

As with other state-of-the-art navigations, the “ChessReader” logo is placed on the left, resembling a home-button that leads back to the index page. If the visitor is not signed in, controls to do so or register a new account are to the right (Figure 33). These change into a “Logout”-button after signing in.

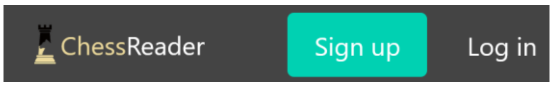


Figure 33: Top Navigation Bar (truncated) – not signed in

For authenticated users their scorecard-management page is available through a link in the navigation bar (Figure 34).

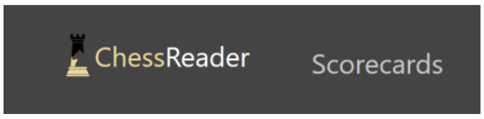


Figure 34: Link to scorecard-management for signed in users

For small screens, mainly mobile devices like phones and tablets, the menu collapses into a “hamburger”-menu, that can be accessed by tapping on it (Figure 35).

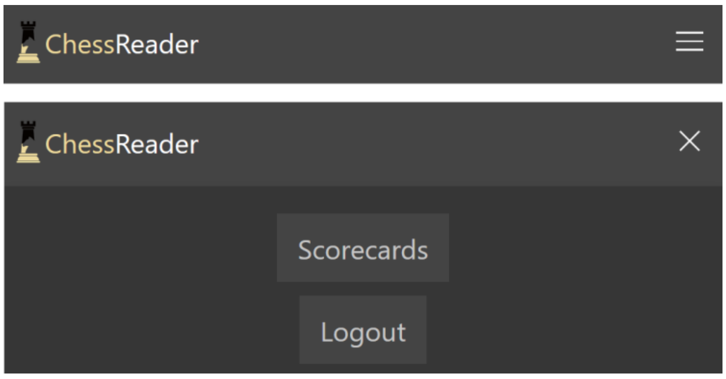


Figure 35: Collapsed and unfolded menu on smaller screens

5.1.4 Usage of Connected Cameras

To further support the idea of only having to use a single device for the whole process, integrated cameras can now be used to capture a live picture of a scorecard directly for uploading (Figure 36).

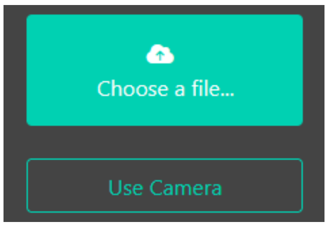
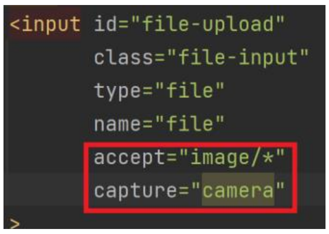


Figure 36: Additional "Use Camera"-button to access a connected camera

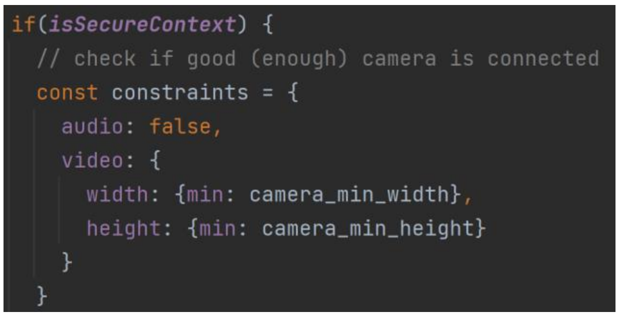
On mobile this is done by using the operating system's default option of letting the user chose to from a gallery or from the camera directly. Only a small change in HTML-attributes is necessary to trigger this behavior (Figure 37).



```
<input id="file-upload"
       class="file-input"
       type="file"
       name="file"
       accept="image/*"
       capture="camera"
>
```

Figure 37: Input-tag attributes for camera semantics

For desktop devices this proved to be a difficult task and required a fair amount of JavaScript programming. First a check for secure context (a secure transport protocol like https being used, or client and server being run on the same machine) is necessary to use the necessary mediaDevice – API. If there is no camera connected to the device, or it does not fit the minimum resolution defined in an environment variable, the option to use the camera is not displayed, since the image quality is insufficient for OCR (Figure 38).



```
if(isSecureContext) {
  // check if good (enough) camera is connected
  const constraints = {
    audio: false,
    video: {
      width: {min: camera_min_width},
      height: {min: camera_min_height}
    }
  }
```

Figure 38: Check for secure context and minimum camera requirements on desktop device

The video capturing device returns a media stream that is displayed in the upper window of a modal. From there a static picture can be taken of which a low-resolution thumbnail is shown in the lower window (Figure 39). If the user is content with the image, the digitalization process can be started using it.

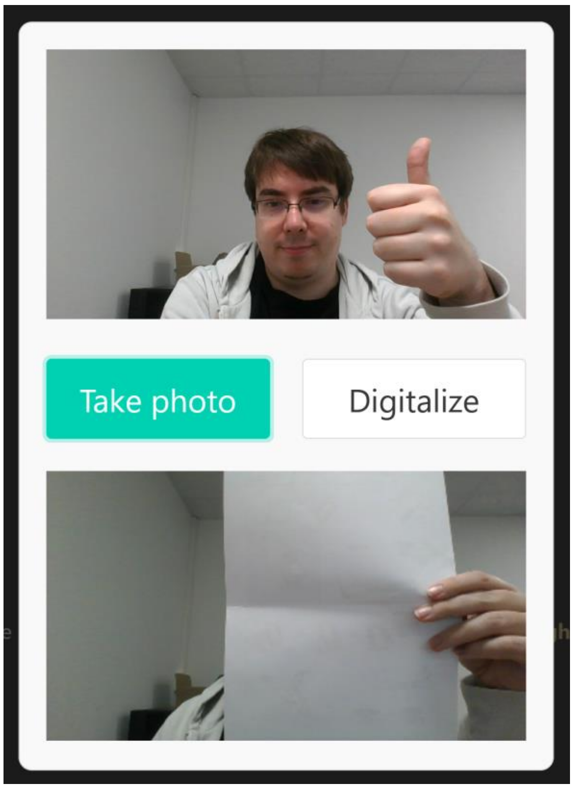
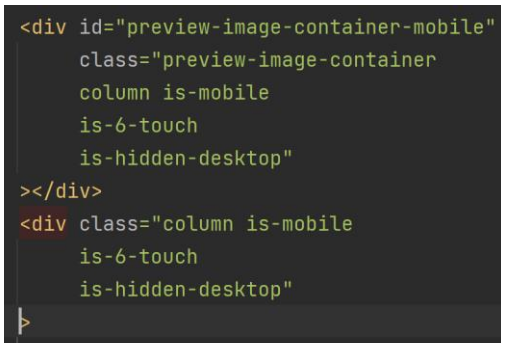


Figure 39: Integrated camera interface for desktop devices showing a live video feed and a static picture

5.1.5 Responsive Design

As defined in the goals for this work (see 1.2 Tasks and Objectives), the application's revised interface had to be responsive to various screen sizes. This was realized using Bulma's responsive column layout and CSS classes [19]. These allow to have fluid column sizes, that can take different amounts of screen space, depending on the screen's size, or even be hidden for certain sizes (Figure 40).



```
<div id="preview-image-container-mobile"
     class="preview-image-container
     column is-mobile
     is-6-touch
     is-hidden-desktop"
></div>
<div class="column is-mobile
     is-6-touch
     is-hidden-desktop"
>
```

Figure 40: Bulma's responsive classes taking care of the styling

The requirements for a mobile (phone) application layout differ from a desktop layout in three core attributes: Screen size, screen orientation and input type. Mobile users generally access websites in portrait orientation, using their right (or left) thumb to access the page's controls. Therefore, all important parts need to be stacked vertically, instead of being spread over a widescreen horizontally. Since mobile devices are usually operated using touch controls, buttons and other controls need to be large enough to be precisely clickable. The much smaller screen does also not allow to have smaller font-sizes. Due to often bad lighting, larger fonts, when compared to desktop devices, are encouraged. Consequently, for the application to fit on the screen without the need to scroll, the interface must be limited to the most crucial parts, while less important ones might need to be hidden or reorganized in menus. Also, primary controls need to be close to the bottom of the screen to be in reach of the user's thumbs, while secondary controls can be on top or hidden in menus, demanding the use of a second hand.

The index page, where scorecards are uploaded, omits the additional hint text about the ideal image resolution, as well as the "Use Camera" button, due to the default behavior described in the previous paragraph "Usage of Connected Cameras" (Figure 41). Unfortunately, the size of blue boxes to confirm the last move on a scorecard are depending on the image size. This did not allow for them to be enlarged to be clickable more easily using touch controls (Figure 42). Like with any website though, they can be zoomed in towards, if the mobile browser used offers this functionality.

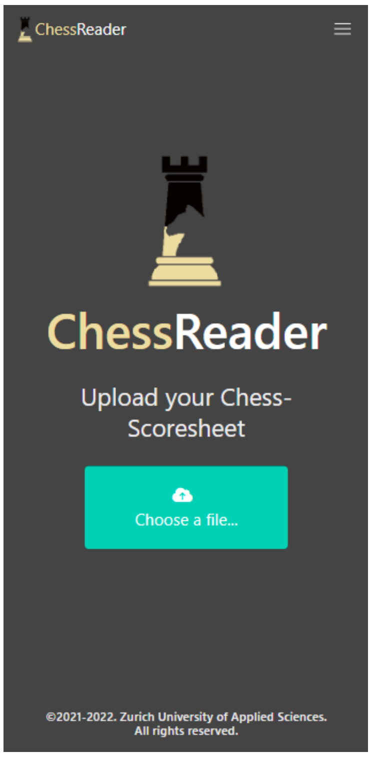

ChessReader
ChessReader
Upload your Chess-Scoresheet
Choose a file...
©2021-2022. Zurich University of Applied Sciences. All rights reserved.

Figure 41: Index-page on a mobile phone device, omitting some text

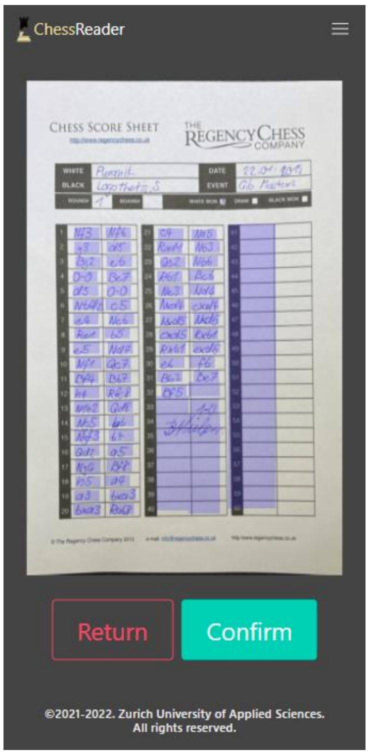

ChessReader
Return
Confirm
©2021-2022. Zurich University of Applied Sciences. All rights reserved.

Figure 42: Buttons to confirm the last box couldn't be enlarged

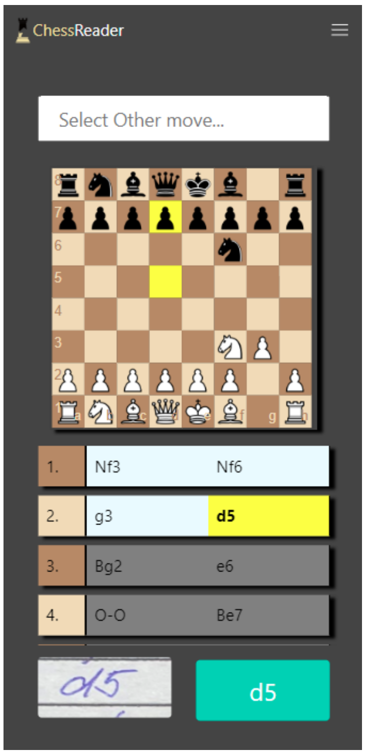


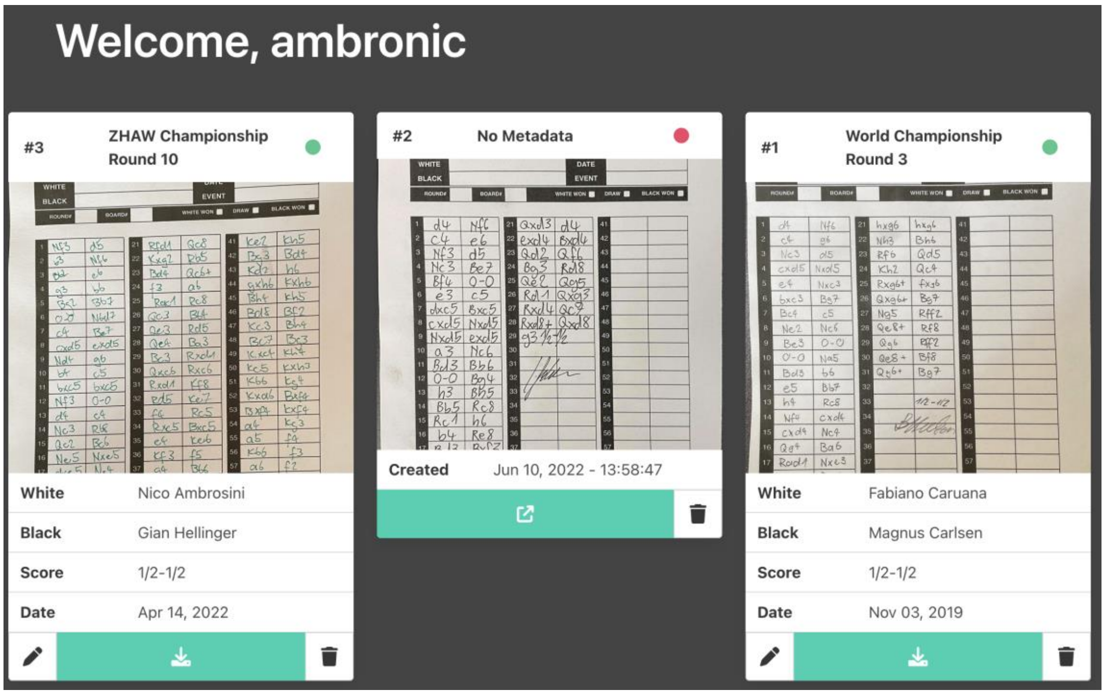
Figure 43: Moves-editor on mobile phone device with reordered controls and less moves in the list

The moves-editor features the most changes, compared to the desktop version (Figure 43). The suggested move and the partial image from the scorecard are shown on the bottom, in reach of the thumbs. The four additional suggestion buttons, as well as the animation controls, are hidden. Therefore, the animation cannot be sped up, but can be paused by tapping the moves list. Navigating through the moves list is also exclusively done by scrolling and tapping into the list, making it the second most important control, having it still located in thumbs' reach. Correcting a move can only be done through the autocompletion on top of the screen. Since this is a secondary action, and the uncollapsing of the list needs some space as well, the bar has been placed at the top, like most similar search functions in other applications.

Once all moves have been validated, the metadata-form appears as normal. After entering and saving them, the PGN-file is automatically downloaded.

5.1.6 Scorecard Management

To give the user the possibility of uploading scorecards and digitalizing them another time, a scorecard management system was developed. A new page was created for the user to see their scorecards (Figure 44).



Welcome, ambronic

#3 ZHAW Championship Round 10

White	Nico Ambrosini
Black	Gian Hellinger
Score	1/2-1/2
Date	Apr 14, 2022

#2 No Metadata

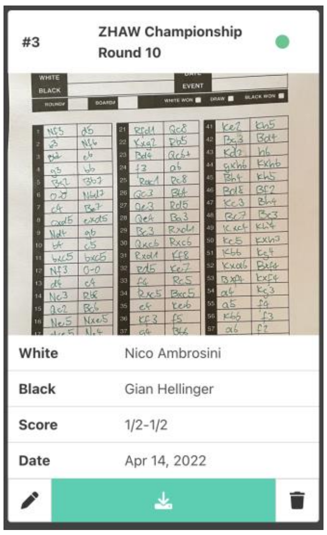
Created	Jun 10, 2022 - 13:58:47

#1 World Championship Round 3

White	Fabiano Caruana
Black	Magnus Carlsen
Score	1/2-1/2
Date	Nov 03, 2019

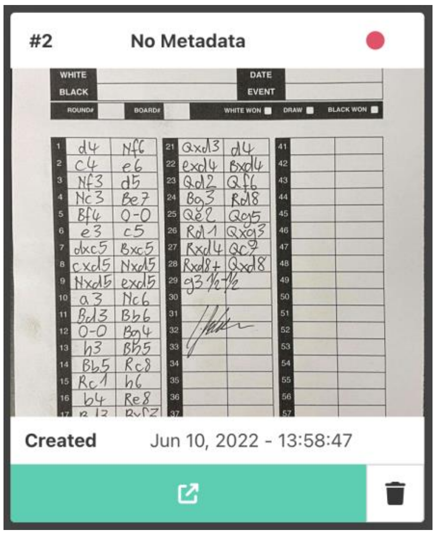
Figure 44: Example of the page showing multiple cards, some with scorecards already digitalized, and one not yet digitalized.

There are two types of scorecards: The one already validated (Figure 45), with the green dot in the top right corner, and those with the red dot, which have not yet been validated. The green scorecards provide three buttons each. The large primary color button signalizes the most important action in downloading the PGN-file. The red scorecards (Figure 46) have only two actions, the primary one being to resume the digitalization process.


#3 ZHAW Championship Round 10

White	Nico Ambrosini
Black	Gian Hellinger
Score	1/2-1/2
Date	Apr 14, 2022

Figure 45: Example of a scorecard that has already been digitalized.

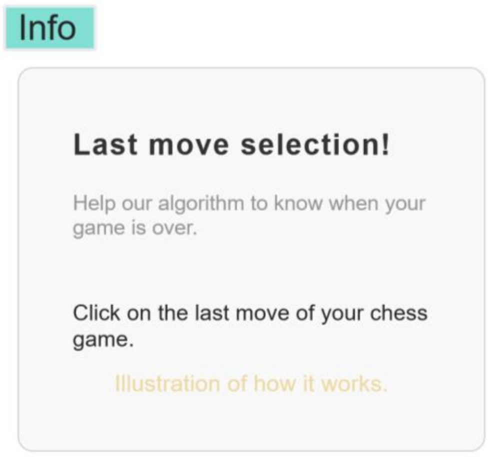

#2 No Metadata

Created	Jun 10, 2022 - 13:58:47

Figure 46: Example of a scorecard that has not been digitalized yet.

5.1.7 Manuals

Previously, hints on how to use the application had been shown when hovering on an “Info” button somewhere on screen (Figure 47). For various reasons, this was not sufficient for “ChessReader”. First, due to the automatic animation in the moves-editor, any user is unlikely to be able to read the information, before the playthrough starts going. Also, the trigger-button would occupy precious screen space, making fitting the GUI on a phone screen even more difficult. Using one-time modals to show the manuals was chosen as an appropriate solution. These pop up over the application, hiding the normal GUI with a transparent background (Figure 48). Once the user has acknowledged the information manually, a cookie is set to stop them from appearing in future uses on the same device and browser.


Info

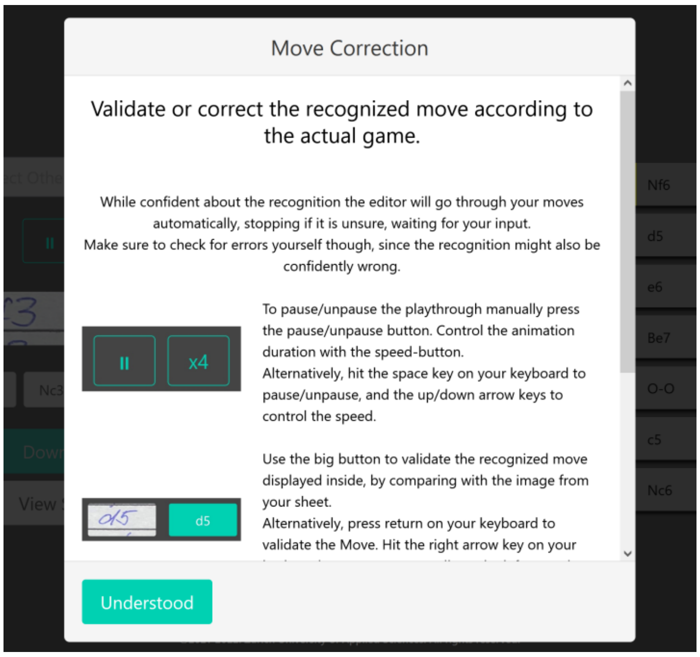
Last move selection!

Help our algorithm to know when your game is over.

Click on the last move of your chess game.

Illustration of how it works.

Figure 47: Tooltips in the original version


Move Correction

Validate or correct the recognized move according to the actual game.

While confident about the recognition the editor will go through your moves automatically, stopping if it is unsure, waiting for your input.
Make sure to check for errors yourself though, since the recognition might also be confidently wrong.

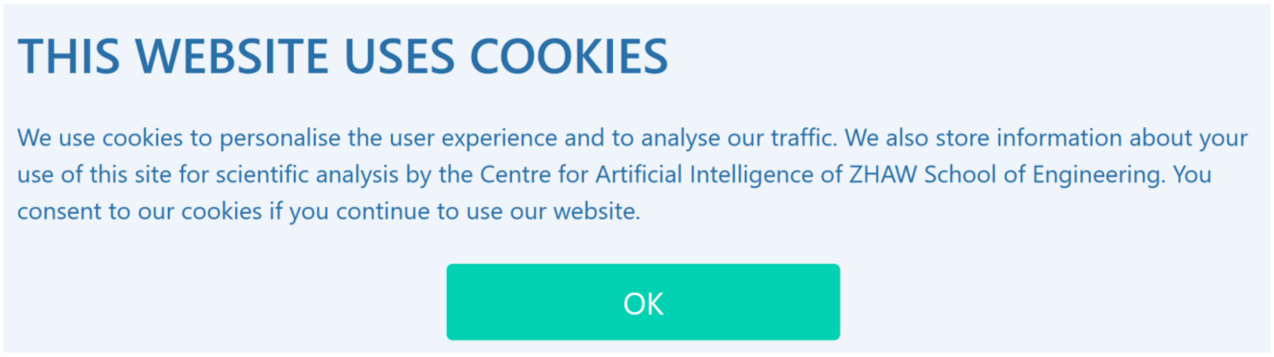
To pause/unpause the playthrough manually press the pause/unpause button. Control the animation duration with the speed-button.
Alternatively, hit the space key on your keyboard to pause/unpause, and the up/down arrow keys to control the speed.

Use the big button to validate the recognized move displayed inside, by comparing with the image from your sheet.
Alternatively, press return on your keyboard to validate the Move. Hit the right arrow key on your

Figure 48: Information modal explaining the moves-editor controls

5.1.8 Cookie-Usage

Some cookies are used by “ChessReader”. Most are to record, if the user has already read – or at least pretended to read – any manuals. Also, one is set to keep the user signed in, if they wished to do so by checking the respective mark when last logging in. A banner informs visitors about cookie usage when opening the application for the first time. As with the manuals, having acknowledged the cookie-banner sets a cookie to not display the banner again.


THIS WEBSITE USES COOKIES

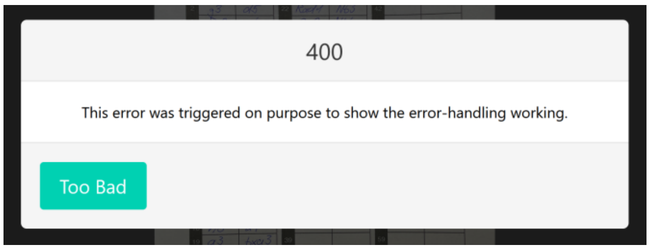
We use cookies to personalise the user experience and to analyse our traffic. We also store information about your use of this site for scientific analysis by the Centre for Artificial Intelligence of ZHAW School of Engineering. You consent to our cookies if you continue to use our website.

Figure 49: Banner informing the user about cookie-usage and information-tracking

5.1.9 Error-Handling

Not even the most well-crafted application is immune to occasional errors happening. Although minimizing them should always be the highest priority, it remains important, as defined in 1.2 Tasks and Objectives, that the user is informed, that something went wrong, and to try again at a later moment.

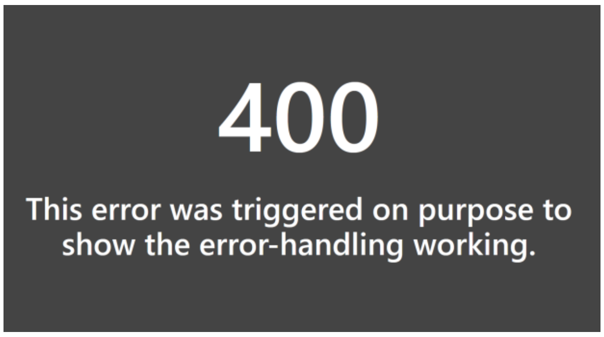
Two different types of error-handling have been implemented, depending on if the request that triggered the error was executed in the background (asynchronously) or not. For an asynchronous error, a modal is displayed (Figure 51), showing the exception's error, then redirecting to the index, while otherwise the user is redirected to the error page (Figure 50).



400

This error was triggered on purpose to show the error-handling working.

Figure 51: Error-modal to handling an asynchronous exception



400

This error was triggered on purpose to show the error-handling working.

Figure 50: Redirection to the error-page if an exception happens

5.2 Backend

The backend of a web application resides on the server, handles data persistence, -access and business logic functions. In the case of “Very Chess”, this mainly included the OCR algorithm and not much more, since the only data persisted were uploaded images, which were saved onto the server’s hard-disk unmanaged. “ChessReader” left most parts of the business logic untouched, except for fixing some bugs. The noticeable changes here lie in introducing a database, a data management layer to organize users, scorecards, and images, and containerization using Docker.

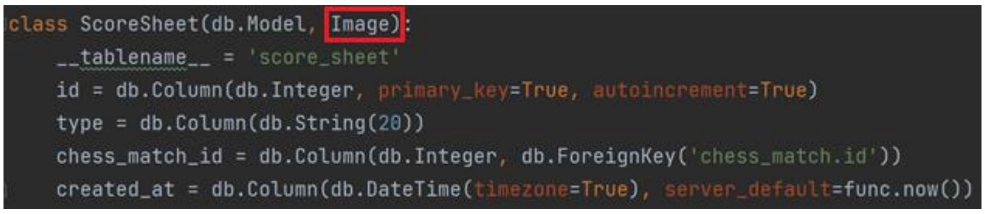
5.2.1 Database

The database consists of six tables (see 9.2 Database Schema): User, ChessMatch, ScoreSheet, MatchMetadata, Move and MoveBoxImage.

The user table is responsible for holding credentials so that any registered visitor can access their resources in the application. The information saved are username, email, and password. For security reasons the passwords are stored hashed, using the sha256 algorithm.

A ChessMatch is created when someone starts digitalizing a scorecard, saving the name of the file for future reference. It belongs to a user and owns a set of MatchMetadata, two ScoreSheets “OriginalSheet” and “AlignedSheet”, and as many Moves as have been validated by the moves-editor. It also stores the response times of external API-calls made during the digitalization process for potential statistics, as well as a Boolean about its overall validation state.

ScoreSheets come in two types: Original and Aligned, the first being the original image uploaded, the second being the result of the align-process-step, where the sheet on the image is properly oriented for the OCR. Each row represents an image stored and managed by SQLAlchemy-ImageAttach, a plugin for SQLAlchemy [20]. Additional attributes are included using the plugin’s Image mixin (Figure 52). While metadata about the images are stored in the database, the actual image-data is saved in the server’s filesystem, organized by ImageAttach.



```
class ScoreSheet(db.Model, Image):
    __tablename__ = 'score_sheet'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    type = db.Column(db.String(20))
    chess_match_id = db.Column(db.Integer, db.ForeignKey('chess_match.id'))
    created_at = db.Column(db.DateTime(timezone=True), server_default=func.now())
```

Figure 52: The ScoreSheet model extending the "Image"-mixin

MatchMetadata stores information about a ChessMatch. Since they are created and saved at a different time than the ChessMatch, as well as using a separate endpoint, they are considered a separate resource and therefore stored in a distinct table.

Moves represent any single move of a ChessMatch. Height, width, x and y describe the position and size of the blue box-button displayed during box-confirmation. They also store the suggested value by the OCR, the actual value after validation, and if they have been manually validated by the user.

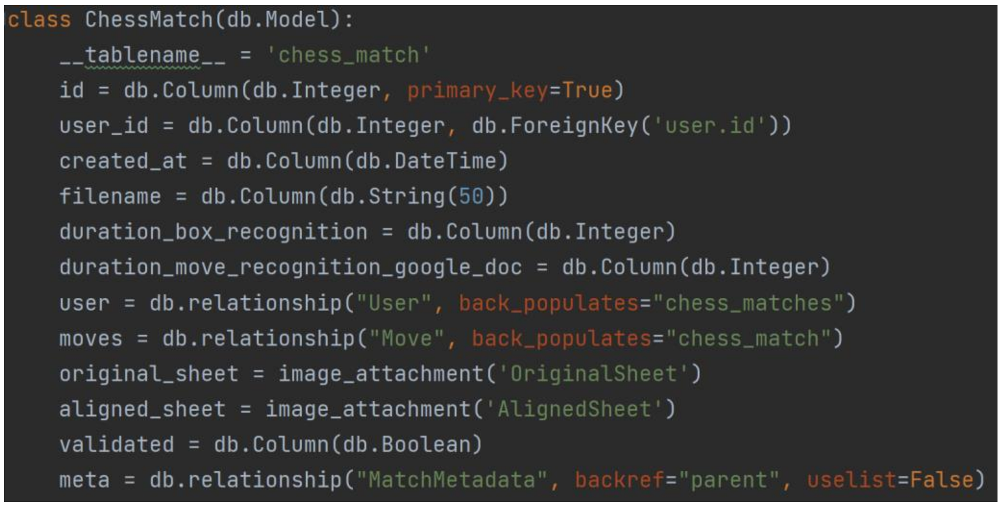
Each move is related to a MoveBoxImage, which is once more extending the Image-mixin of ImageAttach and represents the partial image of the move on the scorecard displayed as a reference during moves validation.

5.2.2 Object-Relational Mapping

To add an abstraction layer between the object-orientated Python code and the relational database, an object-relational mapping (ORM) is used. This makes handling data persistence more intuitive for programmers unused to handling SQL and its different implementations. It also decouples the storage layer, making switching between database implementations, as well as handling remote storages – like on other servers or containers – easier. The ORM used in “ChessReader” is the default used by Flask, SQLAlchemy [21].

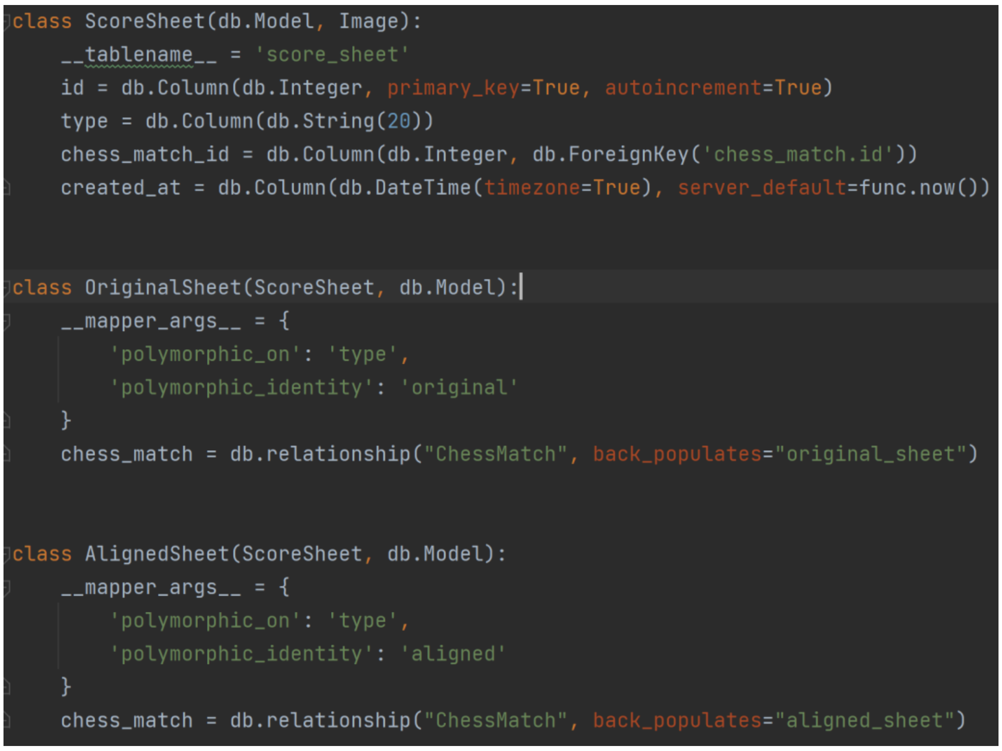
The table structure is defined using entity-classes, which also represent the items in the program code (Figure 53). All the entities that need to be persisted are handed to a relational base (“db.Model” in this case). A shell-script (entrypoint.sh) calls the ORM’s algorithm, which automatically translates the entity-attributes into SQL and creates the database. In development-mode the script is executed every time the web-service is started, while in production it must be executed manually (see 9.3.1 Instructions).

Each ChessMatch has exactly one OriginalSheet and one AlignedSheet, which translates to two one-to-one relationships in SQL. Since they do share all attributes, having two separate tables would be a waste though. This conundrum was solved by using single table inheritance. In the code, OriginalSheet and AlignedSheet are subclasses of the abstract class ScoreSheet, inheriting all its methods and attributes. In the database, they are stored in the same table (ScoreSheet), being distinguished by their value in the “type”-column (Figure 54).



```
class ChessMatch(db.Model):
    __tablename__ = 'chess_match'
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    created_at = db.Column(db.DateTime)
    filename = db.Column(db.String(50))
    duration_box_recognition = db.Column(db.Integer)
    duration_move_recognition_google_doc = db.Column(db.Integer)
    user = db.relationship("User", back_populates="chess_matches")
    moves = db.relationship("Move", back_populates="chess_match")
    original_sheet = image_attachment('OriginalSheet')
    aligned_sheet = image_attachment('AlignedSheet')
    validated = db.Column(db.Boolean)
    meta = db.relationship("MatchMetadata", backref="parent", uselist=False)
```

Figure 53: Parts of the SQLAlchemy-entity defining the ChessMatch-table



```
class ScoreSheet(db.Model, Image):
    __tablename__ = 'score_sheet'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    type = db.Column(db.String(20))
    chess_match_id = db.Column(db.Integer, db.ForeignKey('chess_match.id'))
    created_at = db.Column(db.DateTime(timezone=True), server_default=func.now())


class OriginalSheet(ScoreSheet, db.Model):
    __mapper_args__ = {
        'polymorphic_on': 'type',
        'polymorphic_identity': 'original'
    }
    chess_match = db.relationship("ChessMatch", back_populates="original_sheet")


class AlignedSheet(ScoreSheet, db.Model):
    __mapper_args__ = {
        'polymorphic_on': 'type',
        'polymorphic_identity': 'aligned'
    }
    chess_match = db.relationship("ChessMatch", back_populates="aligned_sheet")
```

Figure 54: Single table inheritance in SQLAlchemy

5.2.3 Container Architecture

To make development and publishing easier, Docker has been used to containerize the application. Two distinct configurations have been created: one for development and the one for production, both done using docker-compose.

Development Environment

In the development environment the application can be run directly using Python. This is bad practice for a Production environment but manageable in a development environment (Figure 55), since the application will use less resources and it automatically adapts to changes made to the code on the fly.

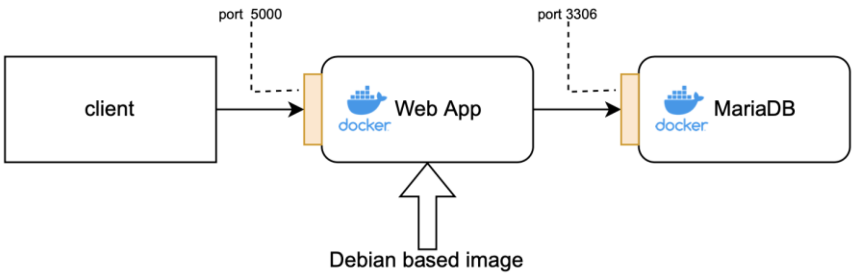


Figure 55: Architecture of the development environment

There are two containers running, one with the web application and the second with an instance of MariaDB. The last is using the official MariaDB image, while the web app is built starting from the official Debian-based Python image and is accessible from the host network on port 5000.

Production Environment

The production environment has quite some difference. In production it is highly recommended to use a Web Server Gateway Interface (WSGI), since Flask it is just a framework and not a web server. It was decided to use gunicorn as the server with nginx as a proxy. There are multiple advantages using nginx in addition to the server:

- It has the capabilities of handling many connections with little CPU and memory usage.
- It can be used to serve static assets, removing some load from gunicorn.
- Nginx makes it easy to switch to the more secure HTTPs protocol to encrypt the data sent between server and client.

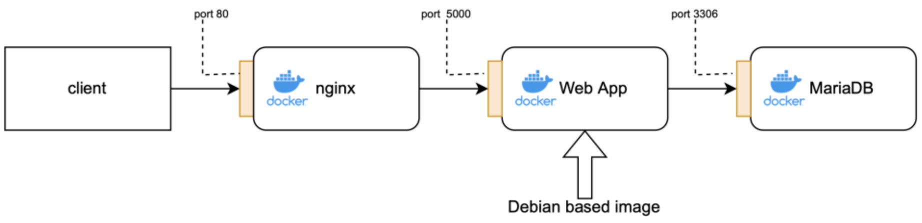


Figure 56: Docker production architecture

The different containers are built and configured as follows:

MARIADB

```yaml
db:
    image: mariadb:latest
    volumes:
      - mariadb_data_prod:/var/lib/mysql
    env_file:
      - ./.env.prod.db
```

The MariaDB container is created using the official image on docker hub [22]. This image is maintained by the MariaDB developer community. A volume pointing to the */var/lib/mysql* folder, where all the databases are saved, is attached to the container, to not lose the contained data each time the container is restarted. The second parameter is an environment file containing the variables to setup the database: **MARIADB_USER, MARIADB_PASSWORD, MARIADB_ROOT** and **MARIADB_DATABASE.**

Web

The container with the web application, requires to be built. This can be done using the following Dockerfile:

```dockerfile
FROM python:3.9.5-slim-buster
RUN mkdir -p /home/app
RUN addgroup --system app && adduser --system --group app
ENV HOME=/home/app
ENV APP_HOME=/home/app/web
RUN mkdir $APP_HOME
WORKDIR $APP_HOME
COPY . $APP_HOME
RUN apt-get update && apt-get install -y --no-install-recommends netcat
default-libmysqlclient-dev build-essential libmagickwand-dev
RUN pip install -r /home/app/web/requirements.txt
RUN chown -R app:app $APP_HOME
USER app
ENTRYPOINT ["/home/app/web/entrypoint.sh"]
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "manage:flask_app"]
```

In short, it tells docker to use 'python:3.9.5-slim-buster' as the base image, creates a user with its home folder, so that the application doesn't run as root, and proceeds to create the relevant folders where the application data resides.

Missing system libraries are installed for the DB connection and image manipulation. Then, the entrypoint.sh script is executed that checks if the DB is online and creates the schema before starting the application. Ultimately, it starts gunicorn binding it to 0.0.0.0, to accept request from outside the host.

Finally, the configuration is added to the docker-compose file:

```
web:
    build:
      context: ./services/web
      dockerfile: Dockerfile.prod
    image: registry.gitlab.com/nambrosini/very-chess/web:latest
    env_file:
      - ./.env.prod
    depends_on:
      - db
    volumes:
      - static_volume:/home/app/web/project/static
      - media_volume:/home/app/web/project/media
      - /home/ubuntu/.chessreader/creden-
tial_store.json:/home/app/web/project/credential_store.json
```

Three volumes are connected to the container, *static_volume* and *media_volume* whlile the *credential_store.json* is a file. This file is critical security-wise since it contains all the keys to access the external APIs for OCR (see 3.3 External APIs). It must not be included in the image to not be available to everyone having access to the image.

Nginx

The nginx container is a proxy to the web container, also serving static assets, reducing load on gunicorn. The second advantage is that HTTPs can directly be implemented in nginx, by generating a certificate, and all the connections passing through it will be encrypted and secure.

Nginx has two configuration files, one for HTTP and the second for HTTPs:

```
server {
    listen 80;
    server_name chessreader.org;

    location ^~/.well-known {
        root /etc/nginx/ssl/bot;
    }

    location / {
        return 301 https://$host$request_uri;
    }
}
```

Two location are defined: the root (‘/’) redirects the users to use HTTPs, while the second (‘**/.well-known**/’) provides the certificates in order to secure the connection.

The second configuration is for securing the connections to the application:

```
server {
    listen 443 ssl;
    server_name chessreader.org;

    ssl_certificate     /etc/nginx/ssl/fullchain.pem;
    ssl_certificate_key /etc/nginx/ssl/privkey.pem;

    client_max_body_size 64M;

    root /home/app/web/project/;

    location / {
        proxy_pass http://web:5000;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $host;
        proxy_redirect off;
    }

    location /static/ {
    }

    location /media/ {
    }
}
```

This defines the path to the certificates, on the nginx filesystem, and what to do when requesting the root (‘/’), static and media files. The root will proxy the connections to the web container where the website resides. A root is defined so that if the client requests assets under the **/media** or **/static** routes, these are directly delivered.

Finally, there is the nginx configuration in docker-compose:

```
nginx:
  image: nginx:1.21.6-alpine
  volumes:
    - static_volume:/home/app/web/project/static
    - media_volume:/home/app/web/project/media
    - /home/ubuntu/.chessreader/nginx/nginx.conf:/etc/nginx/nginx.conf
    - /home/ubuntu/.chessreader/nginx/cache:/etc/nginx/cache
    - /home/ubuntu/.chessreader/nginx/ssl:/etc/nginx/ssl
    - /home/ubuntu/.chessreader/nginx/ssl/bot:/etc/nginx/ssl/bot
    ports:
      - "80:80"
      - "443:443"
    depends_on:
      - web
```

The image is the official nginx alpine version. The ports to open are the standard HTTP and HTTPS ports. It also must depend on the web container, so that it can proxy the requests.

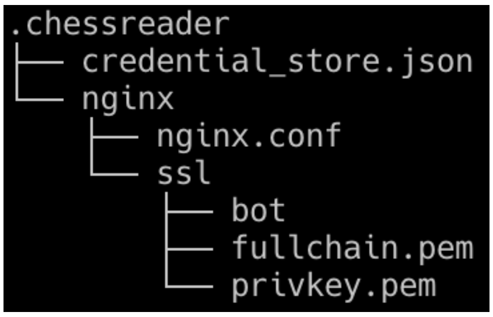
The static and media volumes are the same as in the web container, these will contain the files that nginx has to serve directly. The third is not a volume, but a binding between the configuration file on the host and the container. Nginx will thus use the configuration found on the host. The bottom two are necessary for the SSL configuration.

Server Configuration

The server on which the production version of the application is served, must have the following requirements:

- A working installation of Docker
- Port 80 and 443 must be open

There also needs to be a folder on the host's user home folder, where the following directory containing all the configurations and certificates resides:



```
.chessreader
├── credential_store.json
└── nginx
    ├── nginx.conf
    └── ssl
        ├── bot
        ├── fullchain.pem
        └── privkey.pem
```

Figure 57: Index of nginx home folder

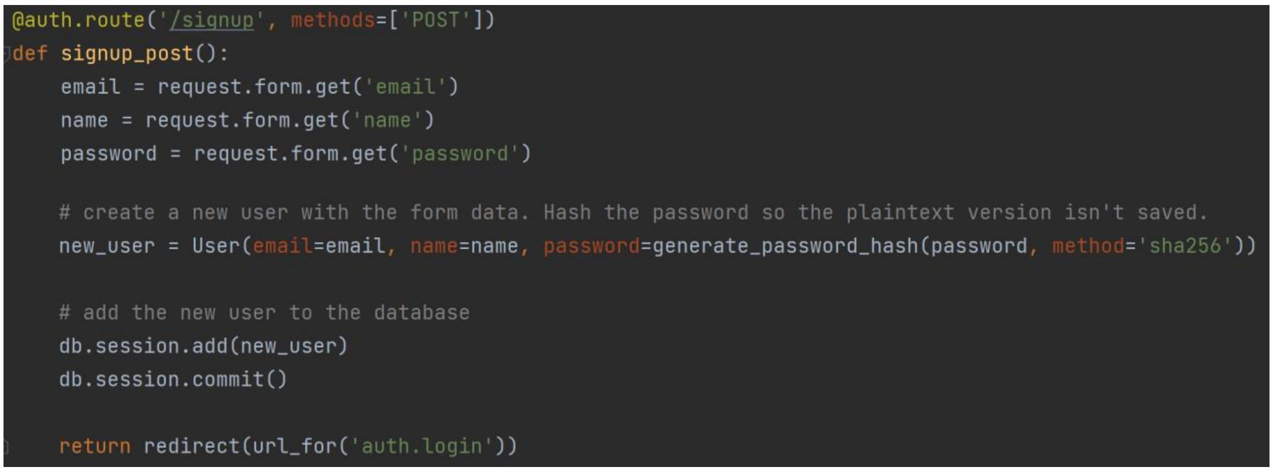
The certificates can be generated using openssl and certbot (for automatic renewal), but for production, these should be issued by an official certificate authority.

5.2.4 User-Login and Scorecards-Management

User authentication and authorization is handled by a Flask-plugin called Flask-Login [23]. It includes a mixin for the user entity which provides methods for authorization, checks for a user to be unique and takes care of hashing and comparing passwords using the sha256 algorithm. It also contains annotations to make selected existing Flask-routes needing the user to be authenticated to access them.

It can also automatically create a blueprint adding new routes necessary for user-management: */login*, */logout* and */signup*. Login and signup respond to either GET or POST requests, either serving the HTML-page rendering the form to each action or handling the provided data as a login/signup demand, accessing the data layer (Figure 58).

The only part of the application currently enforcing user authentication is the scorecards-management (see 5.1.6 Scorecard Management), reachable at */scorecards*.



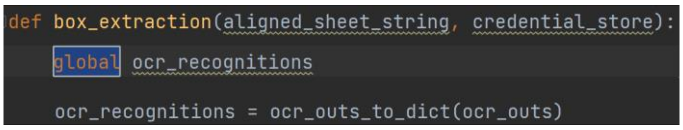
```
@auth.route('/signup', methods=['POST'])
def signup_post():
    email = request.form.get('email')
    name = request.form.get('name')
    password = request.form.get('password')

    # create a new user with the form data. Hash the password so the plaintext version isn't saved.
    new_user = User(email=email, name=name, password=generate_password_hash(password, method='sha256'))

    # add the new user to the database
    db.session.add(new_user)
    db.session.commit()

    return redirect(url_for('auth.login'))
```

Figure 58: Parts of the /signup route (POST) added by Flask-Login

5.2.5 Multiuser-Capability

One of the biggest shortcomings of the prototype was its lack of support for serving multiple users simultaneously. The Python server-script ran on a single thread. Consequently, it could always ever only serve one request at a time. Gunicorn, an HTTP server for UNIX, that spawns and observes multiple worker-processes [24], was used to solve this issue.

Server-client communication also must be stateless to prevent one user's progress of impeding another's. "Very Chess" used global variables though to keep data received from APIs in its memory, using it again in later steps (Figure 59). This would override one user's progress, if another started a new

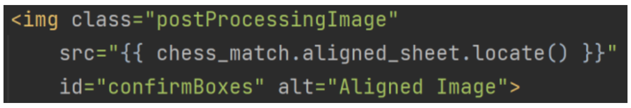


```
def box_extraction(aligned_sheet_string, credential_store):
    global ocr_recognitions

    ocr_recognitions = ocr_outs_to_dict(ocr_outs)
```

Figure 59: The prototype using global variables to store OCR data

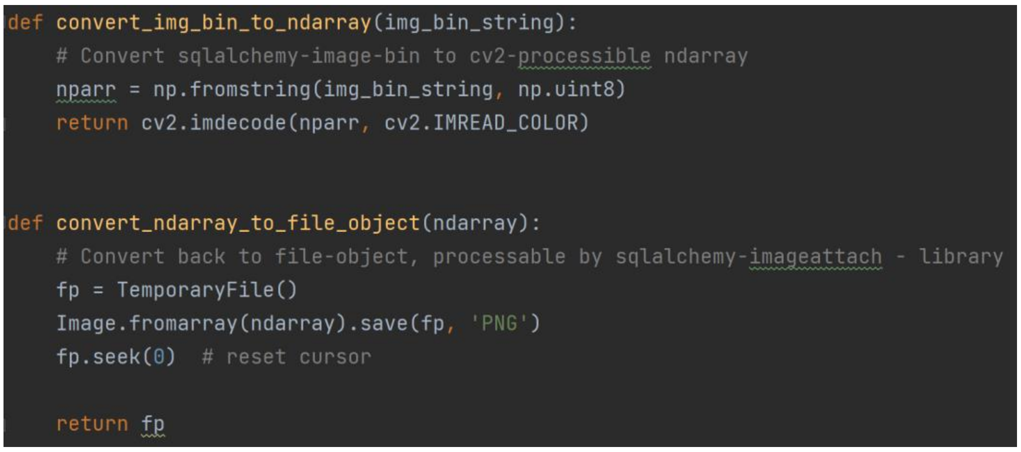
digitalization at the same time, and even make them retrieve wrong results. In addition, if a user's follow-up request would not be served by the same Gunicorn-worker, the data would not be present at all since global variables are not shared between processes (but between threads). Using the database, session-variables, cookies and in one case unfortunately having to make an identic API-call a second time, allowed to circumvent the need for global variables.

Also uploaded files were stored at a hardcoded path in the filesystem, meaning they would over-write each other, if they had the same filename. Using SQLAlchemy-ImageAttach, files are still stored in the filesystem, but the library organizes them in multiple layers of directories, using the related entities primary keys to guarantee unique paths. The plugin serves image-files differently than what some parts of the OCR algorithm are capable to process, having required the addition of transformation methods (Figure 61). To make use of the plugin's *.locate()* method (Figure 60), in development mode it also functions as an HTTP-proxy for the images served, while in production mode exposing image-paths is handled by NGINX (Figure 62).



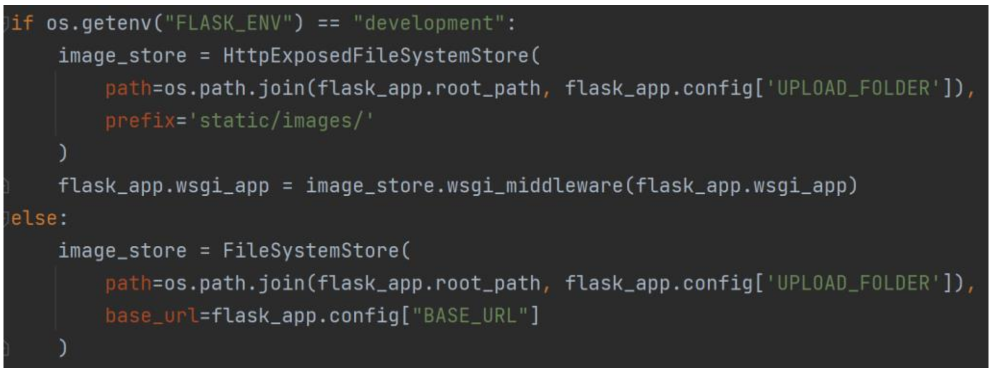
```
<img class="postProcessingImage"
    src="{{ chess_match.aligned_sheet.locate() }}"
    id="confirmBoxes" alt="Aligned Image">
```

Figure 60: ImageAttach's locate() method to insert image-paths in templates



```
def convert_img_bin_to_ndarray(img_bin_string):
    # Convert sqlalchemy-image-bin to cv2-processible ndarray
    nparr = np.fromstring(img_bin_string, np.uint8)
    return cv2.imdecode(nparr, cv2.IMREAD_COLOR)


def convert_ndarray_to_file_object(ndarray):
    # Convert back to file-object, processable by sqlalchemy-imageattach - library
    fp = TemporaryFile()
    Image.fromarray(ndarray).save(fp, 'PNG')
    fp.seek(0)  # reset cursor

    return fp
```

Figure 61: Helper-methods to transform image-objects for different modules to process



```
if os.getenv("FLASK_ENV") == "development":
    image_store = HttpExposedFileSystemStore(
        path=os.path.join(flask_app.root_path, flask_app.config['UPLOAD_FOLDER']),
        prefix='static/images/'
    )
    flask_app.wsgi_app = image_store.wsgi_middleware(flask_app.wsgi_app)
else:
    image_store = FileSystemStore(
        path=os.path.join(flask_app.root_path, flask_app.config['UPLOAD_FOLDER']),
        base_url=flask_app.config["BASE_URL"]
    )
```

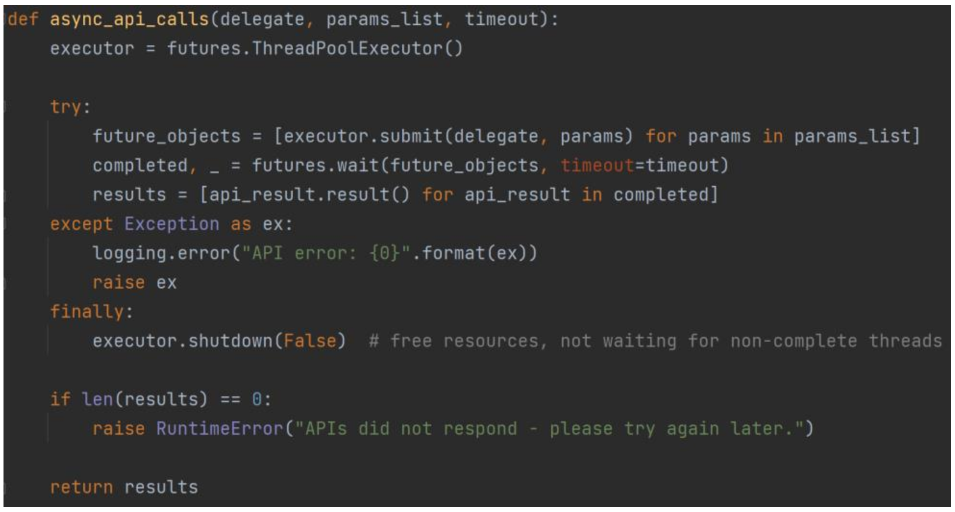
Figure 62: ImageStore configuration for development and production

5.2.6 API Error Handling

“ChessReader” does not feature impactful changes to the core OCR-algorithm when compared to “Very Chess”. Unfortunately, sending and receiving data to/from the various APIs (see 3.3 External APIs) is also the most prone to error segment of the algorithm, since the APIs tend to be unreliable at certain times of days or even whole days at once. This made the application raise server-side errors, aborting the process even though only some of the external services are mandatory to continue the digitalization, while the others only improve the result by comparing recognitions.

As a solution, the algorithm that asynchronously calls all the APIs simultaneously has been changed, to only wait for any answers for a given amount of time, configurable by an environment-variable, before continuing with the responses that arrived in time (Figure 63). The overall quality of the OCR recognition is worse, if not at least three results can be compared, leading to more manual validation being necessary during the moves correction, but the application does not simply abort the process and redirect the user to the error page anymore.

Google’s API data is mandatory for moves recognition though, therefore a missing response there will still lead to an error being risen.



```python
def async_api_calls(delegate, params_list, timeout):
    executor = futures.ThreadPoolExecutor()

    try:
        future_objects = [executor.submit(delegate, params) for params in params_list]
        completed, _ = futures.wait(future_objects, timeout=timeout)
        results = [api_result.result() for api_result in completed]
    except Exception as ex:
        logging.error("API error: {0}".format(ex))
        raise ex
    finally:
        executor.shutdown(False)  # free resources, not waiting for non-complete threads

    if len(results) == 0:
        raise RuntimeError("APIs did not respond - please try again later.")

    return results
```

Figure 63: New method handling asynchronous API-calls with a timeout

5.3 Deployment and Continuous Integration

To help the developers, some automation is taking place. In this case the automation consists in the build and deployment of the application. It is taking place in Gitlab and is triggered when a merge or push is done on either the main or dev branch. The process is divided in two stages. The first stage is the build stage. Here Gitlab gathers all the data needed to build the image, it builds the image and then pushes it in the Gitlab container registry, private to the repository. The second stage is the deployment. Here, based on the branch where the push happened, Gitlab will deploy either on the production server or the development server. Really important are the variable saved in Gitlab under *Settings > CI/CD > Variables.* There are a few variables that must be added:

- CREDENTIAL_STORE: containing the APIs credentials
- ENV_PROD: containing the env variables to use in production
- ENV_PROD_DB: same as ENV_PROD but for the DB
- ID_RSA: the SSH private key to connect to the deployment server. The public key must be added to the authorized_keys file on the server.
- SERVER_IP: the deployment server's IP-address.
- SERVER_USER: the server user where the app will run.

5.4 Bug- and Usability-Fixes

Additionally to the bugs that arose in the first user feedback (see 4.3.3 Bugs) many other problems and usability issues have been fixed (and some new ones implemented and fixed again), including but not limited to:

- Fixed various issues with external APIs, especially ABBYY
- Fixed the last column of a scorecard not being parsed correctly (small, empty blue boxes)
- Fixed removal of duplicated box-recognitions not always working properly (resulting in moves being recognized in-between two lines on the card)
- Fixed a sometimes-occurring infinite loop during box-detection
- Fixed keystrokes still triggering moves-editor actions, while the metadata-form was open
- Set a fixed size for preview-images in the moves-editor for better visibility
- Ordered suggestions in moves-editor autocompletion alphabetically (previous order – if any – was very confusing)

6 Results

Most of the objectives defined at the beginning of the project and after the first feedback (see 1.2 Tasks and Objectives) were direct software requirements. Consequently, a simple acceptance test protocol covering every aim with at least one case was used to determine if they were fulfilled or not. Assessing goals (**G-USE-INT80**) and (**G-PERF-TIMEFF**) was done during the feature and performance evaluation feedback. Video recordings of both the acceptance testing and the second feedback can be found in the “Additional Files”-folder of the hand-in.

6.1 Acceptance Test Protocol

Test-ID	Acceptance Requirement	Execution Details	Test Result
T-USE-BASE	The application allows for digitalizing a scorecard into a PGN-file.		Pass
T-STAB-MULTI	Processing 3 scorecards by 3 different sessions at the same time executes without issues.		Pass
T-STAB-ERRMSG	If an error is raised on any http(s)-request during the process, it is properly shown to the user.	Was tested on index and box-confirmation	Pass
T-USE-RESUME	After the digitalization process of a signed-in user is aborted at a point after box-confirmation, it can be resumed, and the process finishes correctly.		Pass
T-USE-MOBILE	A mobile (phone) device can be used to digitalize a scorecard.	Moves-list did not scroll to current move. Bottom controls sometimes hidden behind action bar.	Pass with potential bugs
T-USE-CHESSB-01	An animated chessboard is displayed when correcting/validating moves, that	Potential bug: Chessboard is flickering with each move on Chrome.	Pass with potential bugs

	plays out the recognized moves automatically, high-lighting the next move. The automatic playthrough can be paused and resumed.		
T-USE-CHESSB-02	When navigating to a given move in the moves-editor, the animated chessboard re-liably shows the board state before that move happened.	Potential bug: Chess-board is flickering with each move on Chrome.	Pass with potential bugs
T-SEC-AUTH-01	A new user can be created and signed in.	Credentials used: tester@testing.test tester 1234	Pass
T-SEC-AUTH-02	Scorecards uploaded when logged in are displayed on a management-page.		Pass
T-SEC-AUTH-03	Scorecards uploaded by a user can only be accessed by that specific user.	Can access any score-card by entering its ID in a given URL!	Fail
T-SEC-HTTPS-01	The server is reachable us-ing the HTTPS-protocol.	Broken HTTPS due to self-signing. Is reachable though.	Pass
T-SEC-HTTPS-02	The server serves a valid HTTPS-certificate.	Untrusted certificate au-thority (self-signed). Cert valid, authority not.	Pass with limitations
T-SEC-COOKIE	An information-banner in-forms the user about the us-age of cookies and tracking.		Pass

Table 4: The finalized application's acceptance test protocol, including results

6.2 Usability

During both feedback rounds, test users were asked, if they thought the application to be “intuitive”. The first version got mixed results, coming down to one “No”, two “Not really/Kinda”s and one “Yes”. These answers correlated directly to the experience level the person claimed to have with OCR/AI applications:

User	Chess-Knowledge [1-10]	OCR/AI – Knowledge [1-10]	First version intuitive?	Second version intuitive?
TU01	7	8	Kinda	Yes
TU02	6	2	No	Better/Yes
TU03	10	10	Yes	Yes
TU04	4	6	Not really	Yes

Table 5: Test group's opinion on intuitiveness of the "ChessReader"

6.3 Performance

During the performance tests, the active time of a user was measured during scanning a scorecard and validating/correcting moves. For the overall time, API response times might influence results to a degree. The performance feedback was executed using scorecards #2 and #3.

User	Scanning [1st | 2nd | diff]			Validation [1st | 2nd | diff]			Overall [1st | 2nd | diff]		
TU01	1:06	0:19	-0:47	3:40	3:23	-0:17	5:36	4:38	-0:58
TU02	2:06	0:15	-1:51	1:03	7:32	+6:29	4:33	8:57	+4:24
TU03	1:10	0:23	-0:47	2:02	5:48	+3:46	4:23	7:25	+3:02
TU04	1:05	0:21	-0:44	2:13	1:35	-0:38	4:14	3:08	-1:06

Table 6: Active time of users while digitalizing scorecard #2 in comparison

User	Scanning [1st | 2nd | diff]			Validation [1st | 2nd | diff]			Overall [1st | 2nd | diff]		
TU01	1:02	0:16	-0:46	1:01	2:47	+1:46	2:41	3:45	+1:04
TU02	1:22	-	-	5:33	-	-	7:51	-	-
TU03	1:31	0:19	-1:12	6:26	2:32	-3:54	8:20	3:59	-4:21
TU04	0:52	0:16	-0:36	3:50	1:00	-2:50	5:28	2:00	-3:28

Table 7: Active time of users while digitalizing scorecard #3 in comparison

7 Discussion

In this chapter the results will be interpreted to judge if the set goals of the thesis have been achieved and what their implications are on future work on and operation of “ChessReader”. Quite some flaws have been discovered during evaluations, methodically as well as feature-wise. These will be discussed in the “Limitations” sub-chapter. Finally, recommendations on what conclusions should be drawn from the experiences gained of this work.

7.1 Interpretation

For each objective set for “ChessReader” (see 1.2 Tasks and Objectives) a verdict is rendered on if they were fulfilled or not, based on corresponding results.

(G-STAB-MULTI)	**Multi-user support for at least three users at the same time.**
Results: 6.1 Acceptance Test Protocol	*Comments:* An acceptance test was performed, accessing the application with three clients at the same time, using different browsers and devices. The test passed without issues.
Verdict: **Fulfilled**	

Table 8: Verdict for goal G-STAB-MULTI

(G-STAB-UPPATH)	**Images are not overwritten due to poor handling server-side.**
Results: 5.1.6 Scorecard Management 5.2.4 User-Login and Scorecards-Management	*Comments:* No specific acceptance test was performed on this goal. The implementation of the SQLAlchemy-ImageAttach library ensures though, that the paths of uploaded images are always unique since it uses their database relations and primary keys to create the folder structure. Since foreign and primary keys guarantee, that a data row in the database can be identified uniquely, the same applies to the file path.
Verdict: **Fulfilled**	

Table 9: Verdict for goal G-STAB-UPPATH

(G-STAB-ERRMSG)	Users are informed about occurring errors.
Results: 6.1 Acceptance Test Protocol	*Comments:* An acceptance test was performed, manually raising exceptions in different parts of “ChessReader”. Asynchronous errors were shown in a modal, redirecting the user to the index page after acknowledging. Synchronous exceptions directly redirected to an error-page, like intended.
Verdict: **Fulfilled**	

Table 10: Verdict for goal G-STAB-ERRMSG

(G-USE-INT80)	80% of test users deem the application “intuitive”.
Results: 6.2 Usability	*Comments:* Asking if the user considers “ChessReader” to be “intuitive” was part of the first and second usability feedback. If the answers given are “rounded” into “Yes” and “No”, for the first feedback only 50% thought it to be so, while after the update 100% of test users agreed, that it can be considered “intuitive”. Going by pure numbers, this leaves the objective fulfilled. Unfortunately, the fact, that only four people were part of the feedbacks, renders this a rather insignificant feat (see 7.3.2 Sample-Size).
Verdict: **Fulfilled**	

Table 11: Verdict for goal G-USE-INT80

(G-USE-RESUME)	An aborted digitalization process can be resumed.
Results: 6.1 Acceptance Test Protocol	*Comments:* An acceptance test was performed, aborting the digitalization before starting it again through the scorecard-management page.
Verdict: **Fulfilled**	

Table 12: Verdict for goal G-USE-RESUME

(G-USE-CAMERA)	A devices' camera can be used to take a scorecard-image directly.
Results: 4.3 Results of First Feedback 5.1.4 Usage of Connected Cameras 6.1 Acceptance Test Protocol	*Comments:* No specific acceptance test was performed on this goal, but scorecards were uploaded using a mobile phone's camera directly during the acceptance test, as well as during the second feedback round, where PC's cameras were (also) used. Using a connected camera on a desktop device is not recommended though, since most devices of that kind nowadays do not provide an image of sufficient quality for the OCR (see 7.3.3 Feature Limitations).
Verdict: **Fulfilled**	

Table 13: Verdict of goal G-USE-CAMERA

(G-USE-MOBILE)	The interface allows for digitalization using mobile devices.
Results: 6.1 Acceptance Test Protocol	*Comments:* An acceptance test was performed, accessing the application with a mobile device. While there were some small bugs and feature-limitations, the digitalization could be completed, and the PGN-file downloaded.
Verdict: **Fulfilled**	

Table 14: Verdict of goal G-USE-MOBILE

(G-USE-CHESSB)	Virtual, animated chessboard plays through recognized moves.
Results: 6.1 Acceptance Test Protocol	*Comments:* Multiple acceptance tests were performed and confirmed, that the chessboard acts as expected. It highlights the next move before playing it out, can be paused at any time and controlled manually.
Verdict: **Fulfilled**	

Table 15: Verdict of goal G-USE-CHESSB

(G-SEC-AUTH)	Users can sign up/sign in to manage and protect uploads.
Results: 6.1 Acceptance Test Protocol *Verdict:* **Failed**	*Comments:* Multiple acceptance tests were performed on this objective. While the database and Flask-Login plugin indeed allowed to create and sign into users to manage scorecards, they were not protected from unauthorized access. This is a major security concern, since personal data might be stored in metadata, as well as partial pictures of a person being uploaded when using a desktop camera to upload a scorecard (see 7.2.1 Data Security).

Table 16: Verdict of goal G-SEC-AUTH

(G-SEC-HTTPS)	The application uses a secure transport protocol (HTTPs).
Results: 6.1 Acceptance Test Protocol *Verdict:* **Partially Fulfilled**	*Comments:* Multiple acceptance tests were performed on this goal. They revealed, that “ChessReader” does indeed use the HTTPs-protocol but serving a self-signed certificate which triggers security warnings in modern browsers. While not being a security concern from the applications point of view, this has the potential to keep visitors from using the application (see 7.2.2 Production Readiness).

Table 17: Verdict of goal G-SEC-HTTPS

(G-SEC-COOKIE)	A disclaimer informs about cookie- and tracking-usage.
Results: 6.1 Acceptance Test Protocol *Verdict:* **Fulfilled**	*Comments:* An acceptance test was performed, showing that an information-banner appears on the bottom of the screen, when the application is accessed for the first time, and it has not been acknowledged yet.

Table 18: Verdict of goal G-SEC-COOKIE

(G-PERF-TIMEFF)	Average time for processing a scorecard is reduced significantly.
Results: 6.3 Performance *Verdict:* **Irrelevant / not enough data to judge**	*Comments:* This objective yielded particularly interesting result. Looking at the results presented, it can be said, that using integrated cameras to directly upload a scorecard objectively speeds up the process by roughly a minute per scorecard or less, depending on how much the file transfer at the beginning would be optimized, or if the image is already on the processing client as a file. Concerning the moves-editor, the results are more sophisticated. Some testers took a lot longer to validate moves with the final version, while others were significantly faster. If the controls provided are used properly, even complex scorecards can be digitalized very fast, using the sped-up animations and keyboard controls, while only judging a valid move by comparing the suggested move with the partial image of the scoresheet (see TU04's results in 6.3 Performance). As observation (**FB-IMO-VALMOV**) showed, most test users – chess-proficient ones especially – heavily focused on the chessboard instead of comparing the suggested move with the image, therefore missing false-positives, if the suggestion made sense on the board state, only realizing with later moves, that something went wrong and having to invest a lot of time into backtracking. As described in 4.3 Results of First Feedback, the backtracking caused by false-positives is by far the most time-intense part of the digitalization process, which is why the rework of the moves-editor is centered around preventing them happening in the first place, instead of just optimizing the OCR-algorithm further, or other performance-based changes. Since the first version did not feature a chessboard, the chess-players had no other option than to just compare suggestions with the scorecard, ironically leading to less false-positives being missed. Nevertheless, the chessboard was the most requested feature by all testers, heavily implicating, that raw performance by means of saving time is arguably not the most significant metric on "ChessReader"s usage (see 7.2.3 Performance), making the premise of the objective flawed in itself. Additionally, it was assured many times during testing,

	that competitive chess-players – who are still the main target audience of the application – would know the moves they made while digitalizing their scorecards, therefore making recognizing false-positives by watching the chessboard a lot more feasible. A set of scorecards of unknown games were used during the feedbacks though, trying to make performance results more comparable by levelling the playing field for all participants. Consequently, this did not represent the most common use-case and arguably rendered the resulting data useless (see 7.3.1 Using A Fixed Set of Scorecards for User Feedback). Ultimately, there is not enough data to issue a final verdict on this objective, while it might also be irrelevant to the applications quality.

Table 19: Verdict of goal G-PERF-TIMEFF

7.2 Implications

The most impactful of the effects of the results gathered in this work on its further development and operation are discussed here, namely the implications about data security, production readiness and performance of “ChessReader”.

7.2.1 Data Security

Multiple outcomes revealed severe security issues. While not generally storing or recording any especially sensitive personal data according to Swiss federal law [25], saving metadata about a “ChessReader”-user’s games, images of their handwriting, as well as potentially storing (partial) pictures of their faces (when using a desktop camera for digitalization) implies significant risks, if abused by potential attackers. Moreover, uniquely identifying personal data – like an image that can be used for facial recognition – are considered especially sensitive personal data in the upcoming privacy law revision [26], most likely coming into effect in September 2023 [27]. “ChessReader” in its current state does not implement any specific security measures, except for those implicitly existing through the infrastructure, tools and libraries used.

7.2.2 Production Readiness

In terms of features, infrastructure, software-architecture, and stability the application can be released to the public with some limitations (see 7.3.3 Feature Limitations and 7.3.4 API-Reliability) and recommendations (see 7.4.1 Operation, Patches and Updates), as mentioned above, security-wise there are major flaws that should be addressed before a potential release, or in the foreseeable future, necessarily before the enactment of the revised privacy law (see 7.4.2 Security Analysis).

7.2.3 Performance

At the beginning of this work, the most important metric for “ChessReader”’s performance was deemed to be the amount of time saved when digitalizing a scorecard, compared to doing it manually. Results on the user feedback have shown, that this is not necessarily the case. The software is basically an application of the “Internet of Everything”, combining people, data, and physical devices to optimize a given process – digitalizing a chess scorecard into a PGN-file –, with the aim of providing additional value to its consumers, in comparison to competing solutions. Not always does this inherent value lie in a simple gain of time efficiency, but more often in providing a personalized experience, a certain look-and-feel or other attributes not directly measurable with statistical tools. This suggests, that for “ChessReader” to be a feasible alternative to manually generating a PGN-file, it matters less, how much time it saves, but more, that the users feel like their time using it is better spent than digitalizing manually. While to some this possibly results in a run for pure time-efficiency, for others it might be more about the overall experience, or having a vault for their scorecards and PGN-files etc. It is therefore highly recommended to conduct additional feedback rounds (see 7.4.4 Large-Scale User Feedback) and/or involving a small amount of proficient players in a shorter, continuous feedback-loop while developing (see 7.4.3 Key Person – Feedback-Loop).

7.3 Limitations

7.3.1 Using A Fixed Set of Scorecards for User Feedback

Chess-proficient users showed similar patterns in handling the application and recognizing the information that is provided by it. E.g., the arrow keys on the keyboard were intuitively used to navigate through the moves of a match, since this is also commonly done on many other chess platforms. Also, they were more focused on the chessboard and the moves being played out. They were automatically validating every move for their meaningfulness in context of the game state.

During the feedback rounds the test-users had to work with scorecards that were handed to them, meaning they neither played the recorded game themselves, nor were they able to judge hard-to-read passages by knowledge of their own handwriting. The combination of this and the usage-patterns led to an artificial increase of time necessary to complete the digitalization process. Errors by the OCR algorithm often slipped through unnoticed, because the recognized move at this point still made sense in the game’s context but made some of the later actual moves illogical or even invalid, therefore a lot of time was spent on backtracking to where the first mistake in the character recognition was present. This would have happened a lot less - or even not at all – if the users digitalized their own played games and was less prevalent for less knowledgeable testers who mainly focused on comparing the picture of the written move with the OCR’s suggestion.

7.3.2 Sample-Size

Another big and more obvious drawback of the observed results is their non-representativeness due to the very small number of test-users. While the quality of the feedback provided by few individuals was exceptionally high, having a serious impact on the features and updates developed, quantitative the size of the test group is far too small to confidently correlate its opinion to an average potential user of the product. Due to their behavior and usage of the application, there is no doubt, that all users are representing the target audience very accurately, nevertheless there are still large differences in terms of skill, approach of the game, experience with similar applications and experience with scorecards and the SAN specifically throughout the target group, which could have been depicted more precise with a larger group of test-users.

7.3.3 Feature Limitations

Feature-wise, “ChessReader” has some flaws and limitations including but not limited to scorecards, notations, and camera-quality.

Scorecard Limitations

During preceding works’ optimizations of the OCR-algorithm, most scorecards used had very similar layouts, if not the same. When using different types, the application started showing weird behavior to some extents, like rotating the scorecard by ninety degrees, subsequently not being able to recognize any moves. Also, the usage of a layout with lower row-heights resulted in additional moves being recognized in-between lines.

While some of these issues have been fixed (see 5.4 Bug- and Usability-Fixes), unexpected behavior can still be expected to happen with different layouts.

Writing Limitations

To have the OCR work at its best, it is necessary that the handwriting is as clean as possible. There are additional problems here though, e.g., some players tend to write timing notes into their scorecards for future analysis. These confuse the OCR algorithm into recognizing them as moves or other issues. Additionally, only the English SAN is currently supported.

Camera Quality on Non-Mobile Devices

While mobile device’s integrated cameras provide impeccable picture quality on very high resolutions, this is not the case for common desktop cameras, which usually support resolutions from 720p up to 1440p. During feature testing, a picture taken with a 1080p camera was not sufficient for the OCR to work correctly. Consequently, the camera feature might only be feasible for very few desktop users.

7.3.4 API-Reliability

The external APIs used for OCR (see 3.3 External APIs) have proven a certain grad of unreliability. Some of them have sometimes very slow response-times for limited periods of time. Especially the ABBYY-API proved to be the cause for many issues during development, due to highly delayed responses, but also connectivity issues and other. Since “ChessReader” was deemed to work perfectly well without the use of ABBYY, the API is not used in the applications final release in this work.

There’s also accounts and traffic limits that must be managed and observed for all of them if a surge in usages of the software should take place.

7.3.5 Code Cleanliness

Due to the various projects being conducted on the same piece of software, featuring a multitude of developers with differing knowledge, experience and diligence, the quality and cleanliness of the code produced varies significantly. This applies, but is not limited to:

- Naming conventions
- Meaningful names for methods and variables
- Library usage (e.g., vanilla JavaScript vs. JQuery)
- Missing commentary
- Excessive usage of confusing language constructs for non-experts (Python)
- Coherency, coupling & redundancy
- Unoptimized excessive nesting of loops

Consequently, the code becomes much harder to read and might increase the time needed to successfully become acquainted with it with every following project.

7.3.6 Automated Tests

So far, no automated tests of any kind (unit, integration, system, etc.) have been written for the business logic of the application. This is probably rather normal for the types of scholarly projects that took place but can be considered a flaw in professional software development, leading to prolonged release cycles and making more debugging necessary.

7.4 Recommendations

Concluding the discussion part, the following adaptations to methodologies and objectives are recommended, based on the experience gained during the process of this work.

7.4.1 Operation, Patches and Updates

No productive version of “Very Chess” or “ChessReader” was running and had to be maintained during this project. Since production-readiness was achieved with limitations (see 7.2.2 Production Readiness), this might be different for the next work, and can be taken advantage of. While the state of a potential productive version must be maintained and occurring bugs and issues fixed in due time, free feedback could also be gathered by productive users, if they are provided the means to do so, e.g., through a contact form.

7.4.2 Security Analysis

As a minimum to increasing data security, measures to prevent unauthorized access to a user’s related data must be implemented as soon as possible. An overall, thorough analysis of the underlying infrastructure, the tools and frameworks used and the application’s code itself in terms of security risks and how to counter them, is advised. Especially regarding the upcoming revision of Swiss privacy law [27], which potentially renders some of the data collected by the application into especially sensitive data.

7.4.3 Key Person – Feedback-Loop

If subsequent works plan to rely on similarly structured feedback, it is hardly recommended to apply some adaptations. It was first planned to have multiple feedback iterations over the whole length of the work, like how it is done in agile software development according to SCRUM and other frameworks. This proved to be difficult in a low workload environment – in comparison to fulltime working schedules that implement SCRUM. Therefore, only two different versions of the software were tested in a late stage of the dissertation. This is due to the thought of having an application that is ready to be tested, before gathering feedback from all test-users in a batch, since the initial version lacked critical features like serving multiple users at the same time.

Opposed to more common scientific Bachelor Thesis research, where questionnaires are usually distributed in large numbers, filled-out unattended and then evaluated statistically once, the feedback gathered in this work took a significant investment of time for every single person, but also directly influenced the development of features and fixes in the software. This feedback, even from only a few test users, proved to be greatly valuable. The profit gained could have been even larger, if at least one person with a vast knowledge of competitive chess would have been included in the process regularly from an early phase on, in an actual loop, regardless of the (feature-wise) state of the program. A broader evaluation process with a larger audience could still take place at a later key-moment in the development process.

7.4.4 Large-Scale User Feedback

As discussed, (see 7.3.2 Sample-Size), the sample-size of the feedbacks conducted was very small and therefore only allowed for highly limited conclusions being drawn, in terms of their applicability to the target audience. A quantitively more elaborate testing group, e.g., a chess club's members, would allow for far more sophisticated results on usability and performance requirements.

7.4.5 Competitor Analysis

Conducting a limited competitor analysis was planned for after the first feedback round concluded, to compare suggested adaptations with other providers' solutions, but was then discarded due to time issues and the sheer amount of detailed feature and change requests gathered. Nevertheless, it is advised to have a look especially at user interface and types of controls that other chess applications (not necessarily scorecard scanners) provide, due to those being assumed to be present by many chess-players. For example, using arrow-keys to navigate between moves and conducting them by drag-and-dropping pieces on a virtual chessboard are widely supported and therefore intuitively defaulted to for many proficient chess-players.

7.5 Ideas for Further Development

Apart from methodical recommendations, there were also a lot of ideas and suggestions for further improvement of the application popping up, some of which are mentioned here.

7.5.1 Chessboard Drag-and-Drop

During feedback most testers asked for being able to correct/validate moves by drag-and-dropping the pieces on the virtual chessboard. While the library used to power the board (see 3.2.7 Chessboard.js) seems to provide means of implementing such functionality, there was not enough time to do so in this work. It is a highly requested feature though and therefore recommended to be addressed in the next project.

7.5.2 Custom Character Recognition Algorithm

Preceding works repeatedly mentioned the training and implementation of a custom-trained CNN (see 2.4 Convolutional Neural Network) as a way to further improve the quality of the character recognition algorithm. A brief attempt of doing this using Amazon Rekognition was conducted, but quickly unveiled, that the amount of directly available data (different hand-written scorecards) was by no means sufficient to provide feasible training for the artificial intelligence.

Since the newly released version of the application stores images uploaded by signed in users, including the recognition's hit and misses, by the time of further development, a large enough set of data for training might have been accumulated, given the release followed regular usage.

7.5.3 Polished JavaScript – Frontend

Using vanilla JavaScript and JQuery, the application's frontend seems to be mimicking a Single-Page-Application, without really being one, since many pages are still completely and directly served by the backend. A full possible project could be to basically rewrite the HTML-serving parts of the software into a fully-fledged JavaScript-framework-powered frontend (e.g., with Vue.js or React.js) and a separate data-serving API-only backend.

7.5.4 Kubernetes

As for now, the application runs directly on docker, without any container orchestration. This is bad practice since the containers must be managed manually. For example, if a container crashes docker doesn't automatically restarts it but it must be done automatically. Listed here are multiple reasons to why a Kubernetes solution should be used instead of plain docker:

- Kubernetes manages automatically multiple replicas of the same container. This allows some load balancing and more importantly reduces the down times. If one containers crash, there are other already up to take the requests and Kubernetes automatically replaces the crashed instance with a new one.
- Sensible information can be stored in a much secure way using secrets, so there would be no need to store files directly on host.
- Kubernetes can scale automatically based on the network and server load, new containers can be created or shutdown with little effort.
- There is no need to have a custom server, instead the application can be run on any Kubernetes cluster with the correct configuration files.

7.5.5 Light/Dark Mode

As for now, the application has only a dark color scheme. As written in chapter 5.1.1, this choice was taken because a dark website is kinder to the eye in darker environment or at night. The problem with it could be that there are users not used to a dark website and when opened in lighter environments makes the website harder to see. A suggestion is to give the user the choice of which color scheme to use, or even better, to change the color scheme based on the user's OS preferences, since all the major OS now support light and dark mode.

7.5.6 Additional Ideas

Since there was no time to write a paragraph for each of the ideas and suggestions that came up during this project, here are some more in a compact list-format:

- Support uploading PGN-files to act as a vault for them
- Support downloading multiple validated games in a single PGN-file
- Prompt metadata (and save them) asynchronously while the OCR is running in the back-ground
- Support other language notations (German, French etc.)
- Implement a function to only upload a scorecard-picture but not start the digitalization pro-cess, people to take pictures on the fly but only digitalize once they get home etc.
- Support uploading a scorecard divided on multiple images

8 Directories

8.1 List of References

[1] "ChessBase - Chess database with analysis," ChessBase GmbH, 2017. [Online]. Available: https://database.chessbase.com/. [Accessed 26 March 2022].

[2] B. Horváth and C. Dreher, "Document Digitization for Chess Scorecards," Zürcher Hochschule für Angewandte Wissenschaften, Winterthur, 2020.

[3] A. Abduli, "Document Digitization for Chess Scorecards," Zürcher Hochschule für Angewandte Wissenschaften, Winterthur, 2020.

[4] V. Caglayan and B. Nielsen, "Digitalisierung von Schach-Formularen mittels Character Recognition Ensembling und Zug-Korrektur," Zürcher Hochschule für Angewandte Wissenschaften, Winterthur, 2021.

[5] Swiss Infosec AG, «Cookie (Consent)-Banner – Die aktuelle Rechtslage,» 1 July 2020. [Online]. Available: https://www.infosec.ch/blog/cookie-consent-banner-die-aktuelle-rechtslage. [Zugriff am 11 June 2022].

[6] Polaris Marketing and Creative, «Important: Google Announced SSL To Become Mandatory for January 2017,» 9 June 2017. [Online]. Available: https://skookummonkey.com/blog/important-google-ssl-mandatory-january-2017/. [Zugriff am 11 June 2022].

[7] 88th FIDE Congress, "FIDE Laws of Chess," 1 January 2018. [Online]. Available: https://handbook.fide.com/chapter/E012018. [Accessed 29 March 2022].

[8] ABBYY, "OCR - Optical Character Recognition explained," [Online]. Available: https://pdf.abbyy.com/learning-center/what-is-ocr/. [Accessed 29 March 2022].

[9] Wikipedia, "Intelligent character recognition," 27 January 2021. [Online]. Available: https://en.wikipedia.org/wiki/Intelligent_character_recognition. [Accessed 29 March 2022].

[10] IBM Cloud Education, "Neural Networks," 17 August 2020. [Online]. Available: https://www.ibm.com/cloud/learn/neural-networks. [Accessed 29 March 2022].

[11] Wikipedia, "Convolutional neural network," 24 March 2022. [Online]. Available: https://en.wikipedia.org/wiki/Convolutional_neural_network. [Accessed 29 March 2022].

[12] Chess.com, "Chess PGN," [Online]. Available: https://www.chess.com/terms/chess-pgn. [Accessed 29 March 2022].

[13] J. Thomas, «Bulma: the modern CSS framework that just works,» [Online]. Available: https://bulma.io/. [Zugriff am 17 June 2022].

[14] Pallets, «Welcome to Flask - Flask Documentation,» 2010. [Online]. Available: https://flask.palletsprojects.com/en/2.1.x/. [Zugriff am 17 June 2022].

[15] I. Stapleton, «Flake8 - Your Tool for Style Guide Enforcement,» 2016. [Online]. Available: https://flake8.pycqa.org/en/latest/index.html. [Zugriff am 17 June 2022].

[16] C. Oakman, «chessboard.js - The easiest way to embed a chess board on your site,» [Online]. Available: https://chessboardjs.com/. [Zugriff am 17 June 2022].

[17] J. Thomas, «Bulma: Colors,» [Online]. Available: https://bulma.io/documentation/overview/colors/. [Zugriff am 13 June 2022].

[18] J. Thomas, «Bulma: Variables,» [Online]. Available: https://bulma.io/documentation/customize/variables/. [Zugriff am 13 June 2022].

[19] J. Thomas, «Bulma: Columns responsiveness,» [Online]. Available: https://bulma.io/documentation/columns/responsiveness/. [Zugriff am 12 June 2022].

[20] M. Hong, «SQLAlchemy-ImageAttach documentation,» 2017. [Online]. Available: https://sqlalchemy-imageattach.readthedocs.io/en/1.1.0/. [Zugriff am 13 June 2022].

[21] M. Bayer, «SQLAlchemy - The Python SQL Toolkit and Object Relational Mapper,» [Online]. Available: https://www.sqlalchemy.org/. [Zugriff am 13 June 2022].

[22] Docker Inc., «Mariadb - Official Image,» 2022. [Online]. Available: https://hub.docker.com/_/mariadb. [Zugriff am 17 June 2022].

[23] «Flask-Login documentation,» [Online]. Available: https://flask-login.readthedocs.io/en/latest/. [Zugriff am 13 June 2022].

[24] B. Chesneau, «Gunicorn - WSGI server,» 2021. [Online]. Available: https://docs.gunicorn.org/en/stable/. [Zugriff am 14 June 2022].

[25] Parlament of the Swiss Federation, «Bundesgesetz über den Datenschutz,» 1 March 2019. [Online]. Available: https://www.fedlex.admin.ch/eli/cc/1993/1945_1945_1945/de. [Zugriff am 17 June 2022].

[26] Bundesamt für Justiz, «BBl 2018 6003 - Bundesgesetz [...] zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten [...],» 9 October 2018. [Online]. Available: https://www.fedlex.admin.ch/eli/fga/2018/2139/de. [Zugriff am 17 June 2022].

[27] Bundesamt für Justiz, «Stärkung des Datenschutzes,» 3 March 2022. [Online]. Available: https://www.bj.admin.ch/bj/de/home/staat/gesetzgebung/datenschutzstaerkung.html. [Zugriff am 17 June 2022].

[28] Docker Inc., "Docker Documentation," [Online]. Available: https://docs.docker.com/. [Accessed 03 April 2022].

[29] Docker Inc., “Overview of Docker-Compose,” [Online]. Available: https://docs.docker.com/compose/. [Accessed 04 April 2022].

[30] Python Software Foundation, "Welcome to Python.org," [Online]. Available: https://www.python.org/. [Accessed 03 April 2022].

[31] Python Software Foundation, "pip PyPI," [Online]. Available: https://pypi.org/project/pip/. [Accessed 03 April 2022].

[32] Chess.com, «Official Score Sheet,» [Online]. Available: https://www.chess.com/terms/chess-score-sheet. [Zugriff am 29 March 2022].

[33] US Chess Federation, “Official Tournament Score Sheet,” [Online]. Available: https://new.uschess.org/sites/default/files/wp-thumbnails/2019/10/Scoresheet.pdf. [Accessed 29 March 2022].

[34] «Flake 8,» [Online]. Available: https://flake8.pycqa.org/en/latest/.

[35] «MariaDB Docker Image,» [Online]. Available: https://hub.docker.com/_/mariadb.

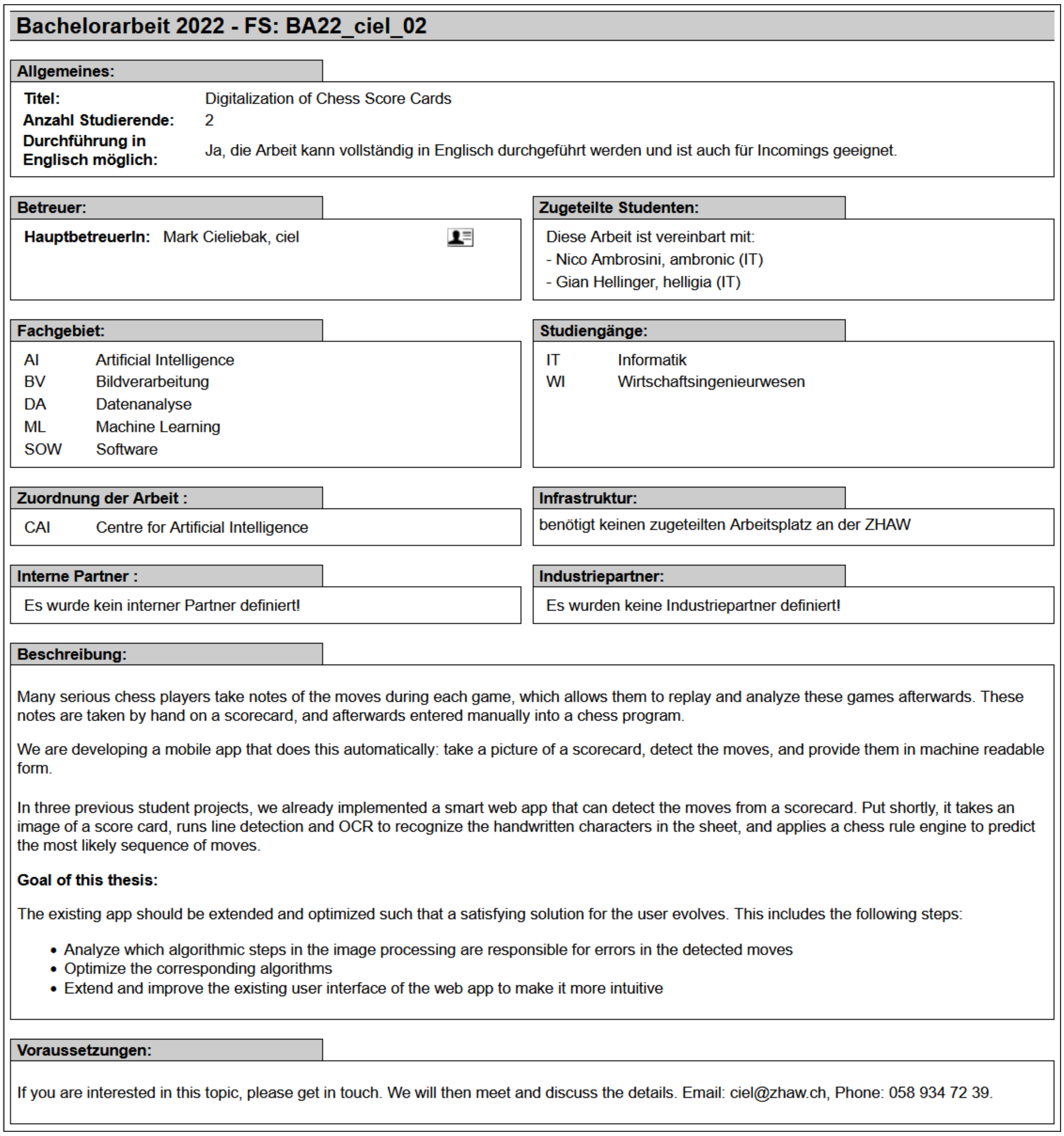
8.2 List of Figures

8.3 List of Tables

9 Appendix

9.1 Project Management

9.1.1 Official Problem Description



Bachelorarbeit 2022 - FS: BA22_ciel_02

Allgemeines:

Titel:	Digitalization of Chess Score Cards
Anzahl Studierende:	2
Durchführung in Englisch möglich:	Ja, die Arbeit kann vollständig in Englisch durchgeführt werden und ist auch für Incomings geeignet.

Betreuer:

HauptbetreuerIn: Mark Cieliebak, ciel

Zugeteilte Studenten:

Diese Arbeit ist vereinbart mit:
- Nico Ambrosini, ambronic (IT)
- Gian Hellinger, helligia (IT)

Fachgebiet:

AI	Artificial Intelligence
BV	Bildverarbeitung
DA	Datenanalyse
ML	Machine Learning
SOW	Software

Studiengänge:

IT	Informatik
WI	Wirtschaftsingenieurwesen

Zuordnung der Arbeit :

CAI	Centre for Artificial Intelligence

Infrastruktur:

benötigt keinen zugeteilten Arbeitsplatz an der ZHAW

Interne Partner :

Es wurde kein interner Partner definiert!

Industriepartner:

Es wurden keine Industriepartner definiert!

Beschreibung:

Many serious chess players take notes of the moves during each game, which allows them to replay and analyze these games afterwards. These notes are taken by hand on a scorecard, and afterwards entered manually into a chess program.

We are developing a mobile app that does this automatically: take a picture of a scorecard, detect the moves, and provide them in machine readable form.

In three previous student projects, we already implemented a smart web app that can detect the moves from a scorecard. Put shortly, it takes an image of a score card, runs line detection and OCR to recognize the handwritten characters in the sheet, and applies a chess rule engine to predict the most likely sequence of moves.

Goal of this thesis:

The existing app should be extended and optimized such that a satisfying solution for the user evolves. This includes the following steps:

- Analyze which algorithmic steps in the image processing are responsible for errors in the detected moves
- Optimize the corresponding algorithms
- Extend and improve the existing user interface of the web app to make it more intuitive

Voraussetzungen:

If you are interested in this topic, please get in touch. We will then meet and discuss the details. Email: ciel@zhaw.ch, Phone: 058 934 72 39.

9.1.2 Planning & Timetable

The period given to fulfil the objectives of this work included 16 weeks, from February 21, 2022, to June 10, 2022.

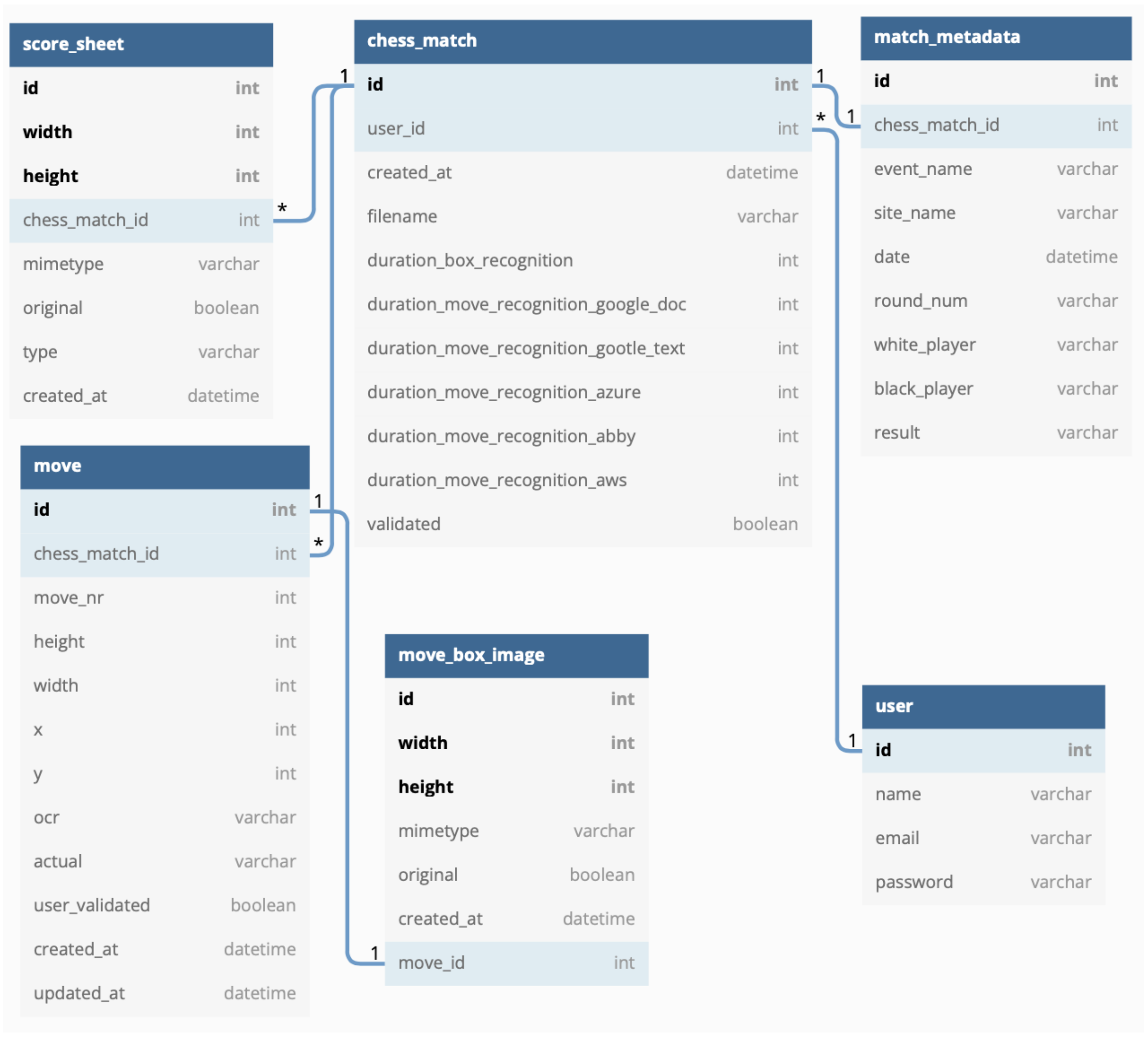
Based on agile project management frameworks, this timespan has been divided into 6 sprints. Except for the first (Sprint 0) and the easter-holiday sprint (Sprint 2) the length of these sprints has been set to 2 weeks each.

Sprint	End-date	Goals
0	March 22, 2022	- analysis of preceding works - define goals - time planning - set-up working deployment - set-up tools & documents
1	April 5, 2022	- user logins - error handling - set-up database
2	April 29, 2022 (*extended because of vacations and sick-ness on April 19*)	- find new name - organize test-user group *(moved from #1 on April 5)* - multi-user capability - user tracking - ~~support scorecards spanning multiple files~~ (*removed on April 5*)
3	Mai 13, 2022	- get initial user feedback *(moved from #2 on April 5)* - usage of device's camera - resumable process & management *(moved from #2 on April 5)* - configure domain
4	Mai 27, 2022	- https - disclaimer - enhance user experience according to feedback - optimization for mobile devices *(moved from #3 on Mai 13)* - ~~read meta-data from scorecard~~ *(moved from #3 on April 5, re-moved on Mai 13)* - ~~performance optimization (2 seconds rule)~~ *(removed on April 5)*
5	June 10, 2022 *(extended to June 17 because of APIs)*	- user evaluation feedback - write report & hand in online - transfer all relevant project-ownerships

Table 20: Planned timetable

9.2 Database Schema

The following image contains the database schema used by the application.



score_sheet	
id	**int**
width	**int**
height	**int**
chess_match_id	int
mimetype	varchar
original	boolean
type	varchar
created_at	datetime

chess_match	
id	**int**
user_id	int
created_at	datetime
filename	varchar
duration_box_recognition	int
duration_move_recognition_google_doc	int
duration_move_recognition_gootle_text	int
duration_move_recognition_azure	int
duration_move_recognition_abby	int
duration_move_recognition_aws	int
validated	boolean

match_metadata	
id	**int**
chess_match_id	int
event_name	varchar
site_name	varchar
date	datetime
round_num	varchar
white_player	varchar
black_player	varchar
result	varchar

move	
id	**int**
chess_match_id	int
move_nr	int
height	int
width	int
x	int
y	int
ocr	varchar
actual	varchar
user_validated	boolean
created_at	datetime
updated_at	datetime

move_box_image	
id	**int**
width	**int**
height	**int**
mimetype	varchar
original	boolean
created_at	datetime
move_id	int

user	
id	**int**
name	varchar
email	varchar
password	varchar

9.3 Miscellaneous

9.3.1 Instructions

Dev-Environment

To start developing locally, a working Docker-backend [28], docker-compose [29], Python 3 or above [30] and pip as a package-manager [31] are necessary.

After cloning the repository to your local drive, install the app-dependencies by navigating to the web-service's root folder and use pip with the requirement.txt – file found there.

```
pip install -r requirements.txt
```

Copy the credentials found in the Gitlab Credentials Store (see 3.2.1 Gitlab) into a new file “credentials_store.json” into the root-folder of the web-service.

Start and build your containers using

```
docker-compose up -d
```

from the applications root-folder.

In development the database is automatically created each time the containers are started. You can – and have to in production – execute it manually, using

```
docker exec very-chess-web-1 python manage.py create_db
```

with the web-service started.