BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

TRANSCRIPTION AND ANNOTATION COMPONENTS FOR WEB EDITOR IN REACT

REACT KOMPONENTY PRO WEBOVÝ EDITOR TITULKOVÁNÍ A ANOTACÍ AUDIA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MATSVEI HAURYLIUK**
AUTOR PRÁCE

SUPERVISOR **Ing. IGOR SZŐKE, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

163530

Institut: Department of Computer Graphics and Multimedia (DCGM)
Student: **Hauryliuk Matsvei**
Programme: Information Technology
Title: **Transcription and annotation components for web editor in React**
Category: Web applications
Academic year: 2024/25

Assignment:

1. Study the React framework and the existing audio transcription and annotation platform. Also study the available transcription tools. Focus on UX/UI.
2. Together with the supervisor, identify appropriate components and features to reimplement and improve the UX of the existing annotation tool. One feature will be collaborative transcription.
3. Design and implement the selected components and features. During implementation, integrate these components and features into the existing interface. Evaluate their UX on a regular basis.
4. Test these components and features on an appropriate group of users. Validate the friendliness and ease of use of the interface.
5. Discuss the goals achieved and suggest directions for further development.
6. Produce an A2 poster and an approximately 30 second video presenting the results of your work.

Literature:

- Tidwell et al.: Designing Interfaces: Patterns for Effective Interaction Design, O'Reilly, 2020
- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN: 978-0321965516, 2013
- Steve Krug: Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability, ISBN: 978-0321657299, 2009
- Joel Marsh: UX for Beginners: A Crash Course in 100 Short Lessons, O'Reilly 2016
- Dále dle pokynů vedoucího

Requirements for the semestral defence:
Items 1-3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Szőke Igor, Ing., Ph.D.**
Head of Department: Černocký Jan, prof. Dr. Ing.
Beginning of work: 1.11.2024
Submission deadline: 14.5.2025
Approval date: 4.4.2025

Abstract

Transcribing and annotating audio recordings efficiently and intuitively is a critical challenge in modern user interfaces. The existing annotation platform often lacks the user experience necessary for seamless interaction, especially in environments with lengthy recordings. This thesis explores ways to improve the existing transcription tool using the React framework, with a focus on improving usability and interface clarity. Several components were created or reimplemented, allowing for a more efficient user interaction. These features were integrated into the existing system and iteratively refined based on regular UX evaluations. The final system was tested with real users to assess its intuitiveness and effectiveness. Suggestions for further development and improvements are also presented.

Abstrakt

Přepis a anotace audia efektivním a intuitivním způsobem představují zásadní výzvu v oblasti moderních uživatelských rozhraní. Stávající platforma pro anotaci často postrádá dostatečně kvalitní uživatelskou zkušenost pro plynulou interakci, zejména v prostředí s delšími nahrávkami. Tato práce se zabývá vylepšením existujícího nástroje pro přepis audia s využitím frameworku React, přičemž důraz je kladen na zlepšení použitelnosti a přehlednosti rozhraní. Bylo vytvořeno či přepracováno několik komponent, které umožňují efektivnější interakci uživatele se systémem. Tyto funkce byly integrovány do stávajícího systému a průběžně vylepšovány na základě pravidelného UX hodnocení. Výsledný systém byl testován se skutečnými uživateli za účelem posouzení jeho intuitivnosti a efektivity. Práce rovněž přináší návrhy na budoucí rozvoj a vylepšení.

Keywords

annotation, communication transcription, audio, subtitles, layout, UI, UX, React, Typescript, metadata, collaborative transcription, cooperative writing, Python, Django, Redux, Wavesurfer, qualitative result, interface testing, entity grouping, metadata, entity tagging

Klíčová slova

anotace, přepis komunikace, audio, titulkování, rozhraní, UI, UX, React, Typescript, metadata, kooperativní přepis, Python, Django, Redux, Wavesurfer, kvalitativný výstup, testování rozhraní, seskupení entit, metadata, označení segmentů

Reference

HAURYLIUK, Matsvei. *Transcription and annotation components for web editor in React.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Igor Szőke, Ph.D.

Rozšířený abstrakt

Tato diplomová práce se zaměřuje na rozvoj a modernizaci webové aplikace pro přepis řeči do textu, která slouží jako nástroj pro lingvistické anotace v rámci projektu JARIN Akademie věd České republiky. Aplikace navazuje na původní řešení vytvořené Jakubem Dugovičem v rámci platformy SpokenData společnosti ReplayWell. Cílem práce bylo zlepšení celkové uživatelské zkušenosti (UX), optimalizace práce s rozsáhlými audiozáznamy a rozšíření funkcionality nástroje tak, aby co nejlépe odpovídal potřebám odborníků.

Namísto vývoje systému od základu se práce soustředí na evaluaci, redesign a systematické vylepšování zděděné platformy. V úvodní fázi projektu probíhá analýza použitelnosti původního řešení, testování s uživateli a komparace s obdobnými nástroji. Paralelně se koná studium literatury zaměřené na návrh uživatelského rozhraní a principy UX, což poskytuje teoretický rámec pro další vývoj a pomáhá lépe porozumět klíčovým aspektům kvalitního designu rozhraní.

Na základě poznatků z testování a zpětné vazby od uživatelů byla navržena a následně implementována celá řada vylepšení. Mezi nejvýznamnější technické přínosy patří zavedení dynamického (lazy) vykreslování segmentů, které zabraňuje výkonovým problémům při práci s velkým množstvím dat; rozšířená možnost úprav metadat a optimalizovaná manipulace se segmenty (např. pomocí drag and drop); vylepšená tokenizace schopná pracovat s nestandardními znaky; modernizace grafického rozhraní; podpora pro kolaborativní anotaci, která umožňuje synchronizaci změn napříč více instancemi aplikace v reálném čase.

Aktualizovaná verze platformy je od ledna nasazena do produkčního prostředí v Akademii věd ČR, kde již byla využita při anotaci několika hodin jazykových nahrávek. Díky vylepšením realizovaným v rámci této práce se významně zefektivnila anotace jazykových dat a zvýšila se použitelnost aplikace pro výzkumné účely.

Probíhající vývoj je již omezen na dílčí opravy chyb a úpravy na základě zpětné vazby. Do budoucna se plánuje doplnění pokročilejší funkcionality a rozsáhlejší testování s cílem kvantifikovat přínosy a interakční chování uživatelů na statistické úrovni.

Dosažené výsledky zahrnují zkrácení doby načítání o **55 %**, zrychlení vykreslení obsahu stránky na téměř konstantní čas (zlepšení o více než **95 %**) a zefektivnění anotace v nekooperativním režimu o **20 %**. Kolaborativní režim navíc zkrátil dobu anotace o **55–65 %**, což výrazně přispělo k efektivitě výzkumné práce.

Transcription and annotation components for web editor in React

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Ing. Igor Szőke Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Matsvei Hauryliuk
May 12, 2025

Acknowledgements

I would like to express sincere gratitude to my supervisor, Ing. Igor Szőke Ph.D., for his guidance and support throughout the duration of the thesis. His advice and feedback have been crucial for my understanding of the subject matter and helped facilitate the development process. I would also like to thank Ing. Josef Žižka for his valuable collaboration with regards to the implementation and technical details. I highly appreciate all the testing participants, especially Mgr. Marta Šimečková, Ph.D. and Grégoire G., for providing vital feedback and sharing valuable ideas for improvement.

I'm beyond grateful to my parents for the unconditional love and support of me in my academic efforts. I salute my friends David Černý and Matyáš Burnek for being there for me throughout my years at FIT.

Contents

Chapter 1

Introduction

Modern software is expected to be intuitive and have a wide range of functionality to support multiple use cases. A crucial factor for the perception of software involves its user interface (UI) and user experience (UX) characteristics. An effective interface can enhance usability, reduce cognitive load, and ensure that users can perform complex tasks with minimal effort. This is particularly important in the field of audio transcription, where accuracy and speed play the key role at scale.

This thesis is dedicated to providing practical improvements for both the graphical interface and the functional components of the existing software platform used for audio annotation of speech recordings. The ATCO2[1] SpokenData platform has also been an essential motivation behind this work. It is used for the editing and processing of air traffic control (ATC) communication recordings.

The ATCO2 project is centered around the collection, organization, and preprocessing of voice data from air traffic communication. The data primarily consists of real-time conversations between pilots and air traffic controllers, captured through very high-frequency (VHF) radio channels or provided by air navigation service providers (ANSPs). To provide additional context, metadata such as surveillance information may be incorporated as well. Such conversations require specialized tools for effective transcription.

Despite this work being inspired by the ATCO2 platform, the scope of the improvements proposed in this thesis extends beyond the domain of air traffic control. The developed platform is designed to satisfy various use cases. The primary use case considered during the creation of this thesis is the annotation of spoken recordings for the Czech language dialect database maintained by the Institute of the Czech Language at the Czech Academy of Sciences. That has greatly influenced the requirements and development goals defined for this project.

At the beginning of the thesis, chapter 2 discusses key UX concepts and evaluates their impact on user interactions. Core directions of UX design in audio transcription platforms are addressed. Special attention is given to the analysis of existing annotation solutions, including ELAN, Audacity, Label Studio, UDT and primarily the system that has been developed as a result of Jakub Dugovič's Bachelor's thesis [17] developed in 2024, which served as a reference point for identifying areas of potential improvement.

Chapter 3 sets the requirements for creating the annotation platform. It also provides detailed use cases to better understand the development goals.

[1] https://www.atco2.org

In Chapter 4 the implementation details are shared. This involves an explanation of the application's architecture, data flow and protocols, highlighting the changes introduced as part of this thesis.

Chapter 5 describes the testing and evaluation process of the implemented features. Qualitative and quantitative methods are used to assess how well the platform meets user expectations and technical requirements. Feedback collected from real-world users plays a key role in validating the proposed improvements.

The final chapter 6 reflects on the thesis results and discusses future development as well as broader implications of the work.

Throughout the development process, the introduced features were continuously tested in real-world scenarios to ensure interface effectiveness. The resulting system strives to provide better technical functionality as well as more valuable user interaction.

Chapter 2

UI/UX and Annotation Platforms

This chapter begins with an introduction of key principles that govern User Interface (UI) and User Experience (UX). These principles not only shape the visual and functional aspects of user interactions with digital systems but also play a critical role in guiding the development and refinement of application layouts. Following this theoretical passage, the chapter provides a comprehensive review of existing annotation platforms. This includes an assessment of their features, usability, and limitations. The chapter is concluded with an evaluation of how these principles have influenced the development process and overall direction of this thesis project.

2.1 UX & UI Development Criteria

This section outlines the core principles of UX/UI design as introduced in foundational literature such as *UX for Beginners: A Crash Course in 100 Short Lessons* [22], *Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability* [19], *Designing Interfaces: Patterns for Effective Interaction Design* [13] and *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability* [20]. These sources made the introduction into the UX/UI discipline and provided a framework for user interface evaluation that would be used throughout the development of the platform.

Prior to implementing any user-facing functionality, it is essential to critically assess the value and purpose of the planned features. This evaluation can be guided by asking the following questions:

- What is the exact **function** of the feature?
- What tangible benefits does the feature offer to the end **user**?
- What is the user's likely next **action** following interaction with this feature?

These estimates help ensure that every design element serves a defined and intuitive role in the user experience process.

2.1.1 Goals of the Development Process

The success of the UX development process is dependent on user psychology, content strategy, usability testing, layout aesthetics, color theory, and emotional engagement. By addressing each of these spheres, developers can influence how well the system is received by the end user.

In order to improve and evaluate the development process, it is important to establish measurable objectives. One of the examples of this are *Key Performance Indicators* (KPIs). These metrics allow to quantitatively evaluate whether a design solution meets its objective. A generalized UX workflow might proceed through the following five stages:

1. Gathering detailed **background** information and context;
2. Conducting in-depth **user research** to understand the target audience;
3. **Designing and prototyping** possible solutions;
4. **Reviewing** the implementation of these solutions;
5. **Measuring** outcomes to assess effectiveness and guide layout refinement.

2.1.2 Understanding User Motivation

Understanding user motivation is central to designing intuitive graphical interfaces. These motivations are often rooted in intrinsic psychological needs of the end users. By addressing these needs, designers can create interactions that feel natural. This not only increases content engagement but also enhances the overall usability and resonance of the application.

2.1.3 Research Techniques

A wide range of qualitative and quantitative research methods are available to gather insights into user behavior and preferences. Some commonly used techniques include:

- **Observation:** Assigning users specific tasks and observing their interactions without interference, followed by post-task interviews.
- **Interviews:** Conducting one-on-one sessions to ask users a structured set of questions.
- **Focus Groups:** Engaging small groups in open discussion. However, caution must be exercised as dominant personalities can sway group opinions, potentially skewing results.
- **Surveys:** Distributing structured questionnaires either on paper or online. The anonymity of this method may encourage more sincere responses, though it lacks the flexibility of follow-up questions and requires careful preparation.

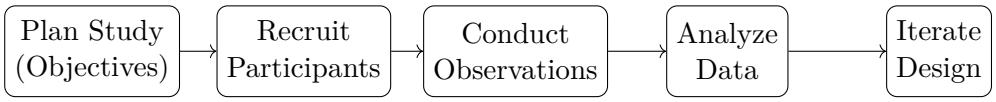

Figure 2.1: Overview of the applied usability study workflow. Within each technique, the objectives are determined at the beginning. Then the participants are gathered. Depending on the method, the participants are invited to answer questions or perform certain actions so that their direct or indirect output can be interpreted. Afterwards, new ideas for the design are introduced accordingly.

The outputs of the techniques are discussed in chapter 5 as part of the description of the testing process. Figure 2.1 illustrates the general workflow for all the testing techniques mentioned.

2.1.4 User Interview Question Types

The effectiveness of user interviews depends on the selection of question types. Each type plays a different role in extracting meaningful insights. **Open-ended questions** invite expansive, narrative responses that can uncover unexpected insights. **Leading questions** guide users in a particular direction and should be used cautiously to avoid bias. **Closed or direct questions** seek specific answers and are useful for clarifying details or quantifying results. Each type of questions serves a distinct purpose in obtaining valuable insights from users. The effective use of these question types contributes to understanding user perspectives, improving the overall quality of user research results.

2.1.5 User Profiles and Personas

Creating detailed user personas is a critical practice in UX design. These profiles serve as realistic archetypes that encapsulate the goals, preferences, and behaviors of different user groups. Effective personas take into account the primary reasons users visit the platform, the information or features they are seeking, as well as potential sources of user dissatisfaction. These insights allow designers to modify the interface to meet specific user needs, creating a more personalized and efficient user experience. The specific user personas considered for the defined use cases are described in section 3.3.

2.1.6 Awareness of User Biases

Cognitive biases are inherent in human decision-making and perception. They represent systematic deviations from rationality and objectivity, often resulting in subjective interpretations of reality. Recognizing these biases is crucial when designing user interfaces, as they influence how users interact with and interpret digital content. Failure to account for such biases can result in user frustration or confusion, undermining the overall effectiveness of the design.

2.1.7 Information Architecture

Information Architecture (IA) refers to the structural design of information environments. As projects increase in scope and complexity, organizing information in a coherent and accessible manner becomes more challenging. IA strategies include the following:

- **Categories:** Grouping content based on shared characteristics.
- **Tasks:** Organizing information based on user tasks or workflows.
- **Search:** Structuring data to support intuitive search and filtering.
- **Time:** Presenting information chronologically.
- **People:** Personalizing content based on user roles or identities.

2.1.8 Key Elements of Layout Design

Effective layout design is essential for communicating hierarchy and guiding user attention. Core concepts include:

- **Line Tension:** Visual continuity that helps users perceive multiple elements as part of a single unit;
- **Edge Tension:** A visual dynamic created by the alignment and interaction of element edges to define shape and structure;
- **Grouping:** Related items should be visually grouped to suggest logical connections without the need for explanatory text.

Figure 2.2: Simulated eye-tracking F-pattern overlay on a news website homepage, illustrating the typical scanning path from top headline, down the left column, and across the bottom row.

Design patterns are recurring solutions to frequent design problems. They can be used to enhance interface usability. One such pattern is the *F-Pattern* (see figure 2.2), which aligns content with the natural scanning behavior of users:

1. Begin scanning in the upper-left corner.
2. Read the top headline or section header.
3. Move vertically down the left side, scanning for keywords or interesting content.
4. Dive deeper into content that catches attention.
5. Repeat the vertical scanning.

This pattern eventually forms an F or E shape and captures the optimal placement of key elements within the layout. By aligning content with this scanning behavior, designers can ensure that the most important information appears in the most visible areas. This not only enhances usability but also ensures that user attention is efficiently directed toward critical content.

2.1.9 User Psyche

One of aims of graphical design creation is to minimize the cognitive burden placed on users. Interfaces should strive to be self-explanatory, reducing the need for users to consciously figure out how things work. For example, links and buttons should appear clickable. Any ambiguity increases mental effort, which can distract users from their primary goals.

Users typically scan pages for familiar cues rather than read content in detail. They look for words or images that align with their immediate needs and tend to settle for the first acceptable option, a phenomenon known as *satisficing*.

To accommodate these behaviors, designers should:

- Leverage established **conventions**;
- Create a clear **visual hierarchy**;
- Break content into clearly defined **sections**;
- Format content to support scanning behaviors;
- Make interactive elements visually **distinct**;
- Eliminate unnecessary **distractions**.

2.1.10 Visual Hierarchy

A strong visual hierarchy ensures that users can instantly grasp the structure of the page. Pages with a clear visual hierarchy have three traits:

- **Prominence**: Key elements stand out via size, color, boldness, or positioning.
- **Logical Grouping**: Related items share a consistent visual treatment or proximity.
- **Nesting**: Visual cues indicate parent-child relationships between components.

Dividing the page into clearly defined areas is important because it allows users to quickly decide which areas of the page to focus on and which areas can be ignored. Users decide in their initial glances which parts of the page are likely to have useful information and then rarely look at the other parts.

Another important way to make pages easy to grasp is to make sure that the appearance of the elements on a page accurately portrays the relationship between them. This is demonstrated on figure 2.3.

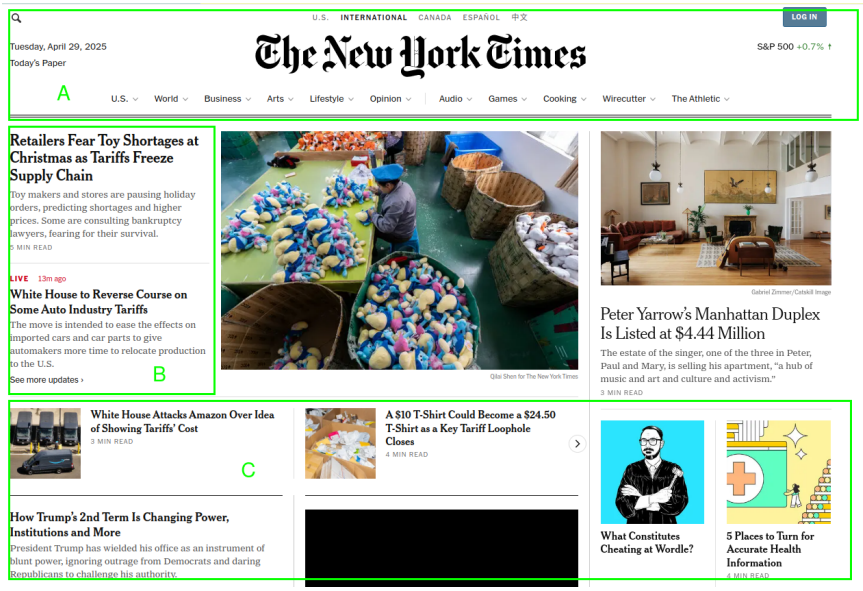

Figure 2.3: Visual hierarchy on a news website. The page is composed of the primary header (section **A**), most relevant stories are situated to the left (section **B**) and the remaining content is placed elsewhere (section **C**).

2.2 Existing Solutions

Before initiating the custom development undertaken for this thesis, an analytical review of existing annotation tools was conducted. This research sought to identify each platform's capabilities, assess their usability, and explore any limitations. Through this comparative analysis, valuable insights were gained which shaped the direction of the UX and UI design process for the custom solution presented in this work. Key characteristics of the reviewed existing solutions are summarized in table 2.1.

2.2.1 Previously Used Platform

The initial version of the application was inherited from Jakub Dugovič's 2024 Bachelor's thesis [17], which tackled a similar task by implementing an annotation platform (see figure 2.4). Although the original solution provided a starting point, its functionality required significant adaptation to meet the needs of the JARIN project. Key functionality has been improved or added to better support the intended use cases, and several new components were introduced to extend the tool's capabilities. In addition, a more consistent interaction design was applied to address usability limitations and enhance the overall user experience.

Several areas for improvement were identified during the evaluation of the initial implementation. These included both functional and usability-related issues that hindered the application's effectiveness, particularly when working with longer audio recordings.

Notable problems identified:

- Inefficient rendering of segments in bulk, which significantly degraded performance and frequently caused the application to freeze on longer recordings.
- Metadata inconsistencies, such as the failure of start and end timestamps to persist after a page refresh.
- Loss of performed changes upon page reload, impacting the reliability of annotation sessions.
- Inability to scroll within the waveform container independently, leading to the entire page zooming in and out instead.
- Playback limitations, including bugs preventing playback speed adjustments.
- Unreliable segment operations, with issues affecting segment creation, resizing, and deletion.
- Limited text editing functionality for segment content.

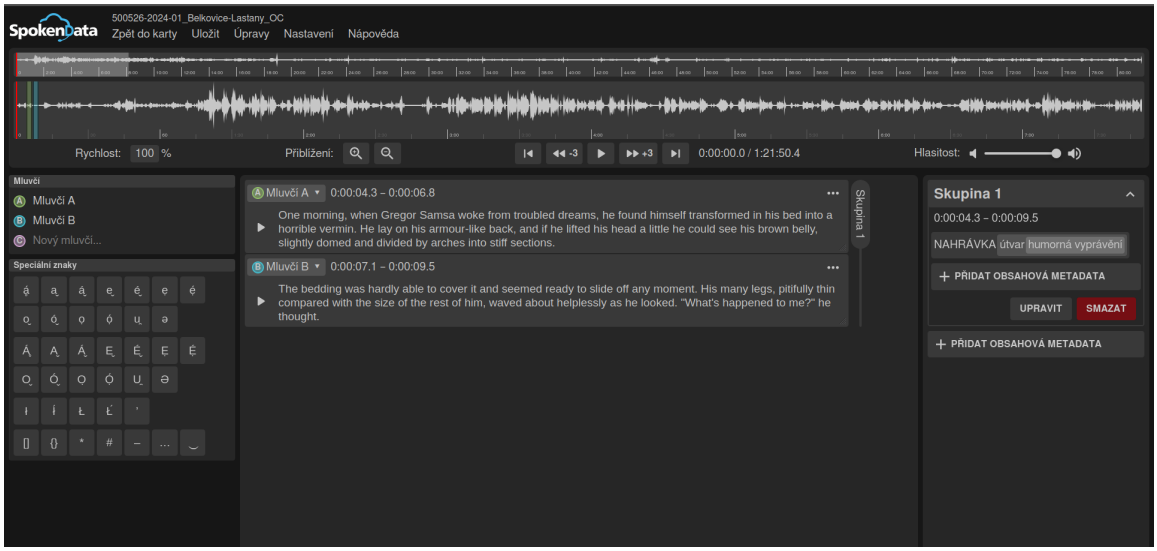

Figure 2.4: Interface of the previously used platform created by Jakub Dugovič.

Additional limitations and usability concerns of the legacy platform will be discussed in more detail in chapters 3 and 4.

2.2.2 Audacity

Audacity[1] is an open-source audio editor that offers a variety of recording and waveform manipulations, effects processing, and audio restoration (see figure 2.5). It supports a broad spectrum of audio editing tasks, from trimming and filtering to applying effects and managing multi-track recordings.

[1] https://www.audacityteam.org

Although Audacity was not specifically designed for annotation tasks, its capabilities make it a useful reference tool when considering basic waveform interaction and editing features. It has an intuitive interface and an extensive toolset.

The advantages of *Audacity* include:

- Precise waveform editing and navigation
- Multi-track audio support
- Wide range of audio processing tools and effects

The missing features of *Audacity* in the context of annotation-focused development include:

- No native support for segment grouping or hierarchical annotation.
- Absence of transcript metadata integration.
- No tools for speaker identification.
- Lack of workflow support for collaborative or web-based annotation environments.

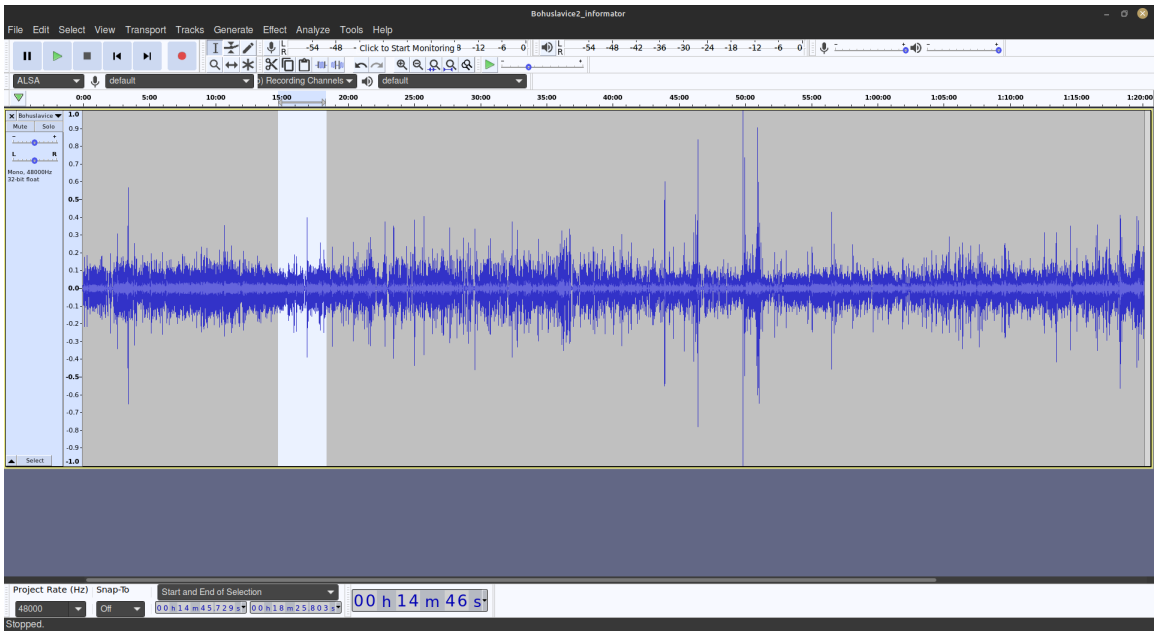

Figure 2.5: Illustration of the **Audacity** audio software interface.

However, despite its strengths as an editor, Audacity was not built with annotation in mind. It lacks fundamental features required for efficient data labeling workflows—such as segment grouping, metadata integration, or dynamic interaction with annotated regions. The absence of speaker separation tools or custom annotation layers further limits its utility in tasks involving structured audio analysis or metadata assignment scenarios.

2.2.3 ELAN

ELAN[2](EUDICO Linguistic Annotator) is a desktop application developed by the Max Planck Institute for Psycholinguistics, designed for the creation of complex annotations on audio and video resources (see figure 2.6). It is frequently used in linguistic and ethnographic research, providing a powerful framework for extensive annotation of spoken language data. ELAN enables users to segment audio manually, organize annotations across hierarchical tiers, and define detailed linguistic types. Annotation data is stored in the XML-based EAF (ELAN Annotation Format), which makes it interoperable with other linguistic tools and formats.

Despite its flexibility and richness in features, ELAN remains largely focused on manual annotation workflows. The application operates as a standalone tool and does not natively support modern collaborative or web-based workflows, limiting its adaptability for large-scale, distributed projects or integration with external metadata databases.

The advantages of the *ELAN* platform include:

- Multi-tiered, hierarchical annotation system.
- Highly customizable tier types and linguistic templates.
- Export options to multiple formats (e.g., CSV, Praat TextGrid, XML).

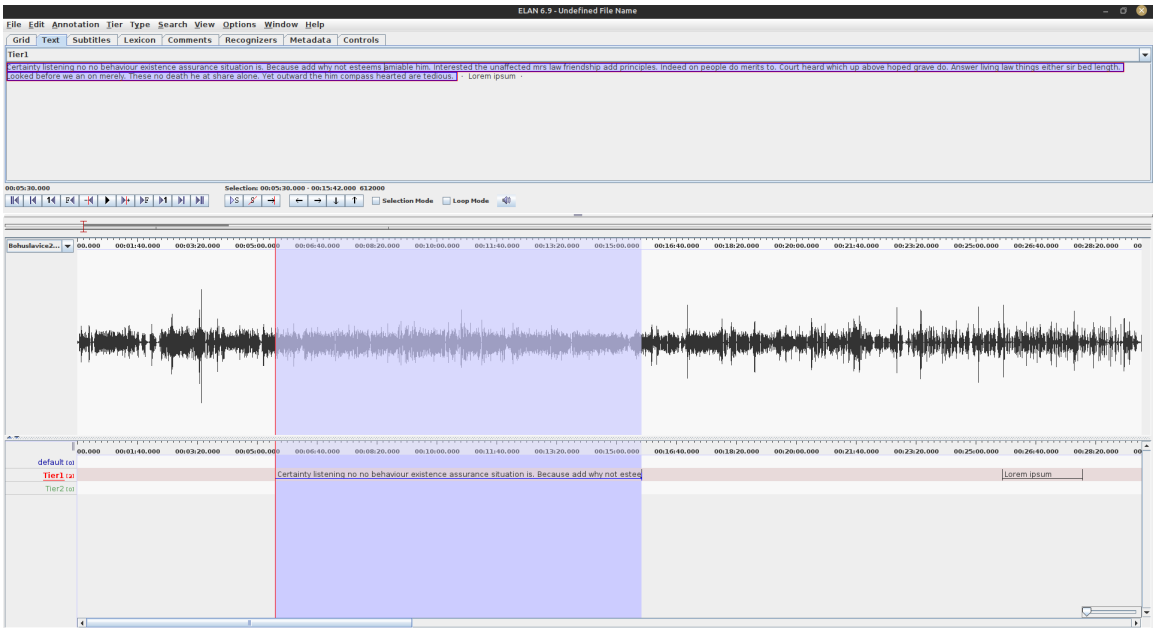

Figure 2.6: **ELAN** linguistic annotator enables audio segmentation, separating recordings into tiers and supports basic annotation.

However, some key limitations of the *ELAN* platform include:

- Manual tier and annotation creation can be time-consuming and unintuitive for new users.

[2] https://archive.mpi.nl/tla/elan

- No built-in support for waveform-based editing enhancements (e.g., **segment grouping**, **lazy loading**).
- Absence of a real-time synchronization mechanism across users or sessions.

In addition, ELAN lacks efficient audio region management strategies. The annotation process in ELAN is designed around precision and depth, but does not prioritize user experience, interactivity, or responsiveness in the way that modern, browser-based tools can. This makes it less suited for high-throughput annotation tasks or rapid prototyping of annotation strategies.

2.2.4 Label Studio

Label Studio[3] is an open-source, web-based data labeling tool developed by Heartex, designed to support a wide range of data annotation tasks across text, audio, images and video. Its customizable interface and extensible backend have led to its adoption in linguistic research. Given the primary use case for this thesis, the most important part of the platform is text segment annotation (see figure 2.7).

Label Studio supports the annotation of audio by allowing users to visualize waveforms, segment regions, and associate metadata or transcriptions with selected audio spans. Annotations are defined using an XML-like configuration system, enabling users to adjust labeling interfaces to the needs of specific tasks or domains. Text labels can be defined manually. Projects can be hosted locally or deployed on remote servers, and multiple annotators can collaborate on the same dataset in real time. Speaker selection when clicking on segments can be viewed on figure 2.8. Collaborative writing implementation within this thesis will be discussed in detail in chapter 4.

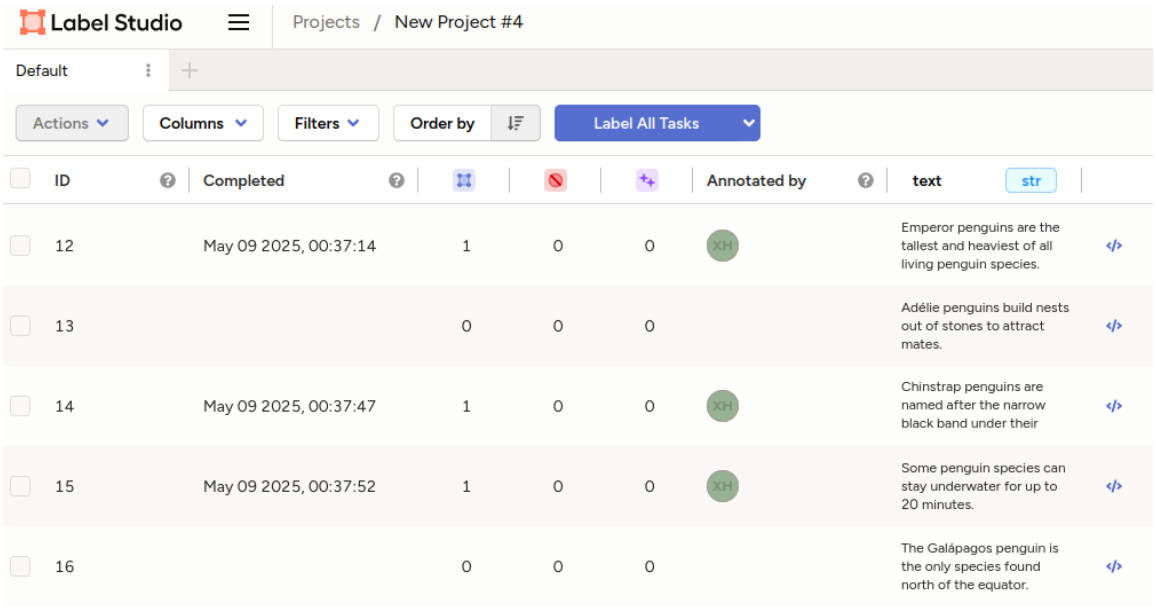

Figure 2.7: Segment annotation within the **Label Studio** platform.

Although highly extensible and better suited for modern collaborative use cases than traditional desktop tools, Label Studio remains limited in its support for advanced annota-

[3] https://labelstud.io/

tion workflows common in linguistic research. Additionally, audio performance for longer files (e.g., 1 hour or more) may suffer unless backend customization or streaming plugins are applied.

The advantages of the *Label Studio* platform include:

- Plugin architecture and REST API integration;
- Native support for waveform-based audio segmentation;
- Web-based interface with real-time collaboration;
- Configurable annotation templates.

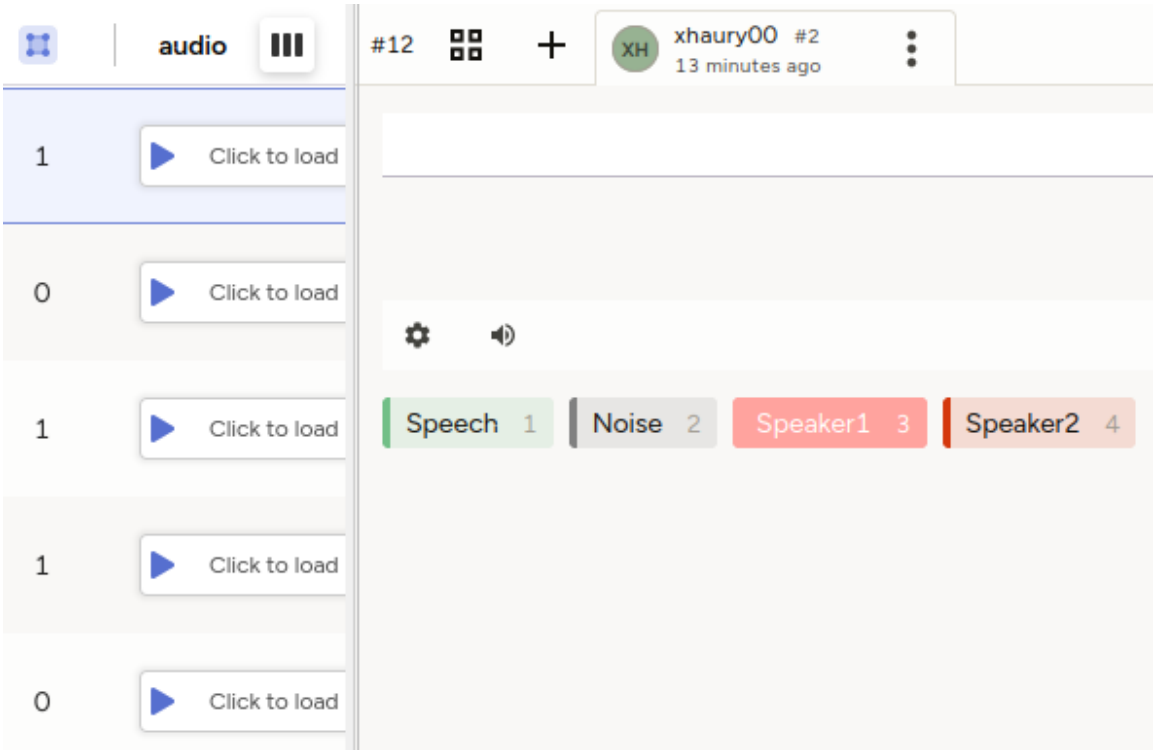

Figure 2.8: Speaker selection in the Label Studio platform.

However, key limitations of *Label Studio* include:

- Performance degradation on long-duration audio files (e.g., >30 minutes);
- Limited support for linguistics-specific data types or formats;
- Requires backend and frontend configuration for optimal audio use cases.

Label Studio places emphasis on appropriate formats of the input data structure. This could pose a challenge for non-experienced users or whenever the data format used by linguists internally does not correspond to Label Studio requirements (e.g., EAF export is not present).

2.2.5 Universal Data Tool (UDT)

Universal Data Tool (UDT)[4] is an open-source, web-based annotation platform designed to support a range of data labeling tasks, including text classification and audio transcription (see figure 2.9). UDT is geared toward rapid prototyping and education, while still accommodating team-based workflows through Git synchronization and shared project hosting.

UDT supports basic audio annotation through waveform visualization, allowing users to segment audio files and assign labels or transcriptions. Annotations are stored in human-readable JSON and CSV formats, which makes the platform accessible for large input volumes.

Collaboration in UDT is primarily asynchronous: the platform supports GitHub/GitLab integration, enabling versioned annotation through pull requests and branching strategies. Additionally, self-hosted deployments offer shared access to projects via URLs, though UDT lacks real time multi-user editing or role based permissions. Unlike Label Studio, it does not provide WebSocket synchronization or advanced backend customization.

Figure 2.9: An illustration of the **Universal Data Tool** interface.

The advantages of the *Universal Data Tool* platform include:

- Web-based interface requiring no installation.
- Support for a variety of data types (text, audio, etc.).

However, the key limitations of *UDT* are:

- No real-time collaboration or live sync.
- Limited support for complex or hierarchical annotations.
- Audio functionality is relatively basic compared to other specialized tools.

UDT is more user-friendly than Label Studio but less customizable than other competitor platforms, which limits annotation abilities.

[4] https://universaldatatool.com

2.2.6 Comparison of Existing Platforms

The existing platforms differ in the scope of features they provide, however none of them is definitively better than others in all aspects, which highlights a need for a new platform. Existing solution differences are comparatively illustrated in table 2.1.

	Legacy	**ELAN**	**Audacity**	**Label Studio**	**UDT**
Performance on 1h Audio	poor	good	excellent	average	average
Speaker Separation	yes	yes	no	no	yes
Segment Grouping	poor	yes	no	yes	yes
Web-based Collaboration	no	no	no	yes	partial
Live Sync / WebSocket	no	no	no	yes	no

Table 2.1: Feature comparison of the legacy platform, ELAN, Audacity, Label Studio, and UDT.

2.3 Conclusion

The reviewed literary sources and existing annotation platforms provided a foundational basis for the development of this thesis, guiding the establishment of clear development objectives and criteria for evaluating interface effectiveness.

Chapter 3

Annotation Platform Requirements and Use Cases

This chapter outlines the foundational motivation behind the thesis and provides a description of the functional and technical requirements for the annotation platform. It begins by introducing the reasons for undertaking this project, followed by a comprehensive listing of the graphical user interface (GUI) features necessary for supporting the platform's intended workflows. Additionally, representative use cases are described to ground these features in practical application scenarios. The chapter concludes with a summary evaluation to guide the design and development stages that follow.

3.1 Motivation for the Thesis

The primary motivation for this thesis arises from the need to develop an annotation platform for the *JARIN* project, an initiative of the Institute of the Czech Language at the Czech Academy of Sciences. The project involves extensive work with spoken language data, requiring a specialized tool for the efficient annotation and transcription of long-form audio recordings.

Although a preliminary version of such a platform existed, it was determined to be insufficient in several key areas that impacted both user experience and workflow efficiency.

Shortcoming	**Impact**
No support for long audio	Poor performance with large audio files, limiting project size.
Poor metadata handling	Difficult organization and retrieval of recording information.
Absent segment tagging	Metadata interaction is limited.
No collaborative editing	Multiple users cannot work simultaneously.

Table 3.1: An estimate of limitations of the existing platform as well as their impact.

Therefore, the following improvements were identified as essential (see table 3.1):

- Support for **extended audio** recordings, including performance optimization and waveform generation.
- **Persistence** of the last annotation timestamp to streamline workflow continuity.

- Robust mechanisms for **tagging** at the word and segment level.
- Enhanced **metadata handling** and contextual integration.
- Better audio navigation, including the ability to focus on specific audio sections.
- Support for **collaborative editing**, enabling multiple users to annotate the same material.

Throughout the development lifecycle, relevant components from existing annotation platforms were studied and selectively adapted. Then, the components were refined and re-imagined to better serve the needs of the project.

As development progressed, each new component was evaluated in terms of usability and functional contribution to the overall application usability. These evaluations were further validated through structured testing sessions involving actual users. Feedback from these sessions played a critical role in iterative design improvements.

In chapter 5 of the thesis, the extent to which the project objectives were met is assessed, and the potential for future development is discussed.

3.2 Required GUI Features

The development process was guided by an initial mockup (see figure 3.1) provided by the project supervisor. This mockup helped define the expected scope and visual design of the application, serving as a starting point for feature planning and implementation.

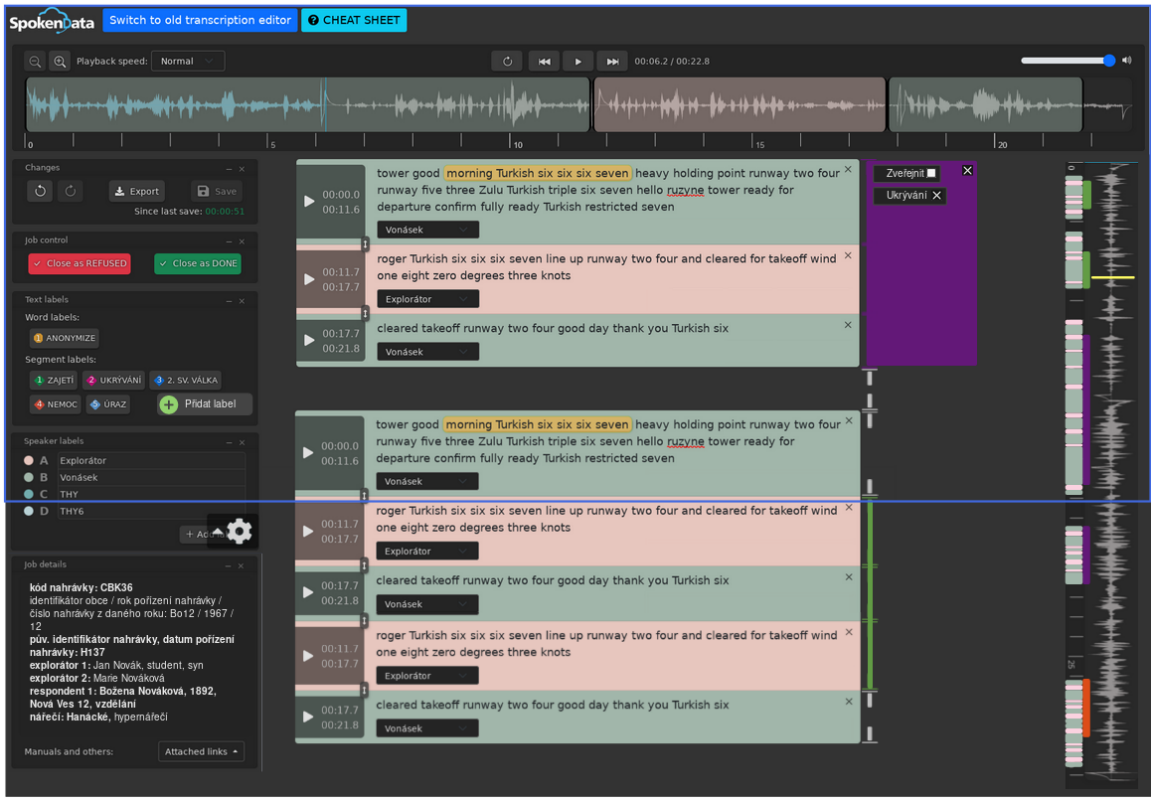

Figure 3.1: **Preliminary design mockup** of the annotation platform.

3.2.1 Generic GUI Requirements

The baseline functionality expected of the GUI includes the following:

1. Automatic **saving** of work to prevent data loss.
2. Change tracking and version **history** for auditability.
3. Undo/redo capabilities to support experimental editing.
4. Display of recording **metadata (title, description, date/time, etc.)**.

Category	**Feature Description**
General GUI	Autosave, undo/redo functionality, metadata display.
Waveform Interaction	Zooming, playback speed control, segment highlighting.
Transcription Editor	Create, move, merge segments; assign speaker identity.
Tagging	Word-level and segment-level tagging; batch editing support.
Audio Player	WAV playback, seeking, timestamp display.

Table 3.2: Required GUI features determined at the beginning of the implementation.

3.2.2 Waveform Component Requirements

The waveform visualization and interaction component is central to the platform and must support:

- Playback and visualization of long WAV files.
- Efficient rendering of pregenerated waveforms for large recordings.
- Dynamic zooming for precision navigation.
- Real-time indication of the current playback position.
- Seeking to specific points via mouse interaction.
- Visual representation of transcription segments.
- Adjustable segment boundaries for accurate timing.
- Highlighting of the currently active or selected segment.
- Variable playback speed control.

3.2.3 Segment Transcription Editor

The transcription editor is expected to facilitate the annotation workflow. It must allow users to create new segments and remove those that are either incorrect or redundant. Adjusting the timing and editing the textual content of segments should be seamless, as should the ability to assign speaker identities to individual segments. If several parts of the annotation need to be logically united, users should be able to merge adjacent segments. The interface should clearly highlight the currently selected or active segment, making navigation intuitive. To improve efficiency, the integration of keyboard shortcuts for frequently used operations is also essential.

Figure 3.2: Typical user workflow for editing audio recordings.

For effective linguistic analysis, the tag editor should support various types of metadata operations at the segment level. Users must be able to create and remove segment groups through straightforward graphical interactions. The system should accommodate segment tag selection, enabling flexible annotation suited to various research needs. A typical audio recording interaction workflow can be viewed on figure 3.2.

3.3 Use Cases

During the mockup and design phase, the development process leveraged the concept of user personas (described in section 2.1.5) to model typical user interactions. These were used to illustrate how real users might engage with different features in practical scenarios.

A representative use case for this thesis is a linguistic specialist interacting with an audio recording to produce a precise transcript of a language sample.

The core objective of the platform developed within this thesis lies in its deployment as an annotation environment for the *JARIN* project. The platform is intended to support linguists and researchers in transcribing various regional Czech speech samples with a high degree of accuracy and annotation richness. This contributes to broader linguistic goals such as dialect preservation, language documentation, and phonetic analysis.

Users are able to annotate recordings with customized segment groups, allowing for a detailed representation of the speech data. The interface is designed for simplicity and accessibility, enabling efficient navigation, interaction, and data entry even for users without technical backgrounds.

Overall, the platform functions as a comprehensive tool for spoken language research and serves as a vital component in digital processing for linguistic data. It also plays a crucial role in the preservation and documentation of regional Czech dialects, supporting long-term linguistic research and cultural heritage efforts.

3.4 Conclusion

Key use cases and program requirements have been carefully identified, providing a clear framework for the implementation process. This structured approach ensured that the platform's development aligned closely with the needs of its intended users and the objectives of the project.

The features prioritized for implementation include support for long audio files, segment operations and word tagging mechanisms.

Chapter 4

Proposed Solution and Implementation

This chapter presents the technical solutions created for the existing platform. In relation to chapter 2, a set of ideas is presented, focusing on the introduction of new features, addressing potential bug fixes, improving system architecture and implementing performance optimizations. These actions aim to improve the overall functionality and user experience of the platform.

4.1 React and Redux

The React framework provides a component-based architecture for building UI elements, while TypeScript adds static typing to JavaScript, enabling better tooling, early error detection, and improved code readability. Redux complements React by providing a predictable state container that centralizes application state management.

In Redux architecture, *types* are used to define the shape and structure of the data being managed within the application. They provide a form of schema that ensures state consistency after the actions that modify it. *Selectors* are utility functions responsible for extracting specific pieces of information from the global state. They help isolate and encapsulate access to particular parts of the state, making the application more maintainable and modular.

The *reducers* are functions that determine how the application's state should change in response to a given action. They take the current state and an action as input and return a new state based on the action type and its associated payload. Reducers must be predictable and free of side effects, ensuring that the same inputs always produce the same output.

Once the state is updated by a reducer, the new state is propagated to all components that are subscribed to the *Redux store*. These components re-render as necessary to reflect the changes, enabling hierarchical data flow and predictable UI behavior. [9, 12, 10, 15]

4.2 Improvements to Existing Components

Key interface evaluation concepts discussed in section 2.1.8 have been considered throughout the implementation to gauge the effectiveness of layout changes. Additionally, feedback received in appendix A has been continously addressed.

4.2.1 Dynamic Segment Rendering

From the earliest stages of using the previous version of the annotation editor, it quickly became clear that the system was not optimal when it came to handling longer audio recordings, especially those exceeding an hour in length/containing hundreds of segments. While working with more extended audio files, various limitations of the annotation editor surfaced – ranging from poor performance when loading transcript segments and unresponsive visual elements to difficulties in navigation and segment management. This highlighted that the application's design and implementation were not optimized for large-scale or extended annotation sessions. Although it functioned adequately for shorter clips or isolated segments, the experience noticeably deteriorated as the length of recordings increased. This realization introduced the need for a more efficient solution.

`loadVisibleRegions` function within the `useLoadRegions` hook has been altered to improve the efficiency of rendering audio segments. The core objective of this modification was to ensure that only the segments currently visible to the user were rendered. The remaining segments are thus rendered on demand, e.g. in case of viewport adjustment, segment creation, segment deletion etc. By avoiding the rendering of all segments at once (and returning anything offscreen), especially for longer recordings, the application gains a considerable performance boost. The measurements related to this claim are shown in 5.5. This is crucial when dealing with extensive files containing hundreds of segments, which can otherwise cause severe performance degradation if all segments are rendered simultaneously.

```
const loadVisibleRegions = () => {
    const visibleRangeStart = wavesurfer.current?.getCurrentTime() || 0;
    const containerWidth = wavesurfer.current?.getWrapper().clientWidth || 0;
    const duration = wavesurfer.current?.getDuration() || 0;
    const pixelsPerSecond = containerWidth / duration;
    const visibleRangeEnd = visibleRangeStart + containerWidth / pixelsPerSecond;

    segments.keys.forEach((key) => {
        const segment = segments.entities[key];
        // Render segment only if it's visible and hasn't been rendered yet
        if (segment.start < visibleRangeEnd && segment.end > visibleRangeStart &&
            !renderedSegments.current.has(key)) {
                const region = waveformRegionsRef.current.addRegion({
                    start: segment.start, end: segment.end, drag: false, minLength: 0.1,
                    color: rgba(speaker2color[segment.speaker] || "#c6c6c6", 0.4),
                });
                dispatch(mapRegion2Segment({ segmentID: key, regionID: region.id }));
                renderedSegments.current.add(key); // Mark segment as rendered
            }
    });
};
```

In its updated form, `loadVisibleRegions` executes a series of checks and actions to manage segment rendering more efficiently. Initially, it loads only the regions that are currently visible based on the user's scroll or playback position. The logic begins by determining the start and end of the visible range, using values such as `visibleRangeStart` (based on the current playback time), `containerWidth` (the width of the waveform container in pixels), and `pixelsPerSecond` (which converts pixel values into corresponding time durations). With these metrics, the `visibleRangeEnd` is calculated, effectively defining the boundaries of the portion of audio that is visible at any moment.

Once this range is known, the function iterates through all available segments and determines whether each one falls within the visible time window. If a segment is within view and has not already been rendered, it is then added to the transcript. This conditional rendering ensures that each segment is only added once, preventing duplicate renders and unnecessary overflows.

Additionally, segments that are outside the visible range are skipped from rendering (see figure 4.1), ensuring the visual structure remains intact while reducing rendering load.

To process dynamic interactions, such as user-initiated scrolling, zooming, or seeking in the waveform, corresponding util functions handle these events. The functions are invoked whenever the waveform view is updated by such interactions. The visible range is recalculated and a call to `loadVisibleRegions` is triggered, ensuring that newly visible segments are rendered on demand. This dynamic loading mechanism is tied to key `WaveSurfer` events like `region-update-end` and `zoom`, which the application listens to in order to determine when the view has changed.

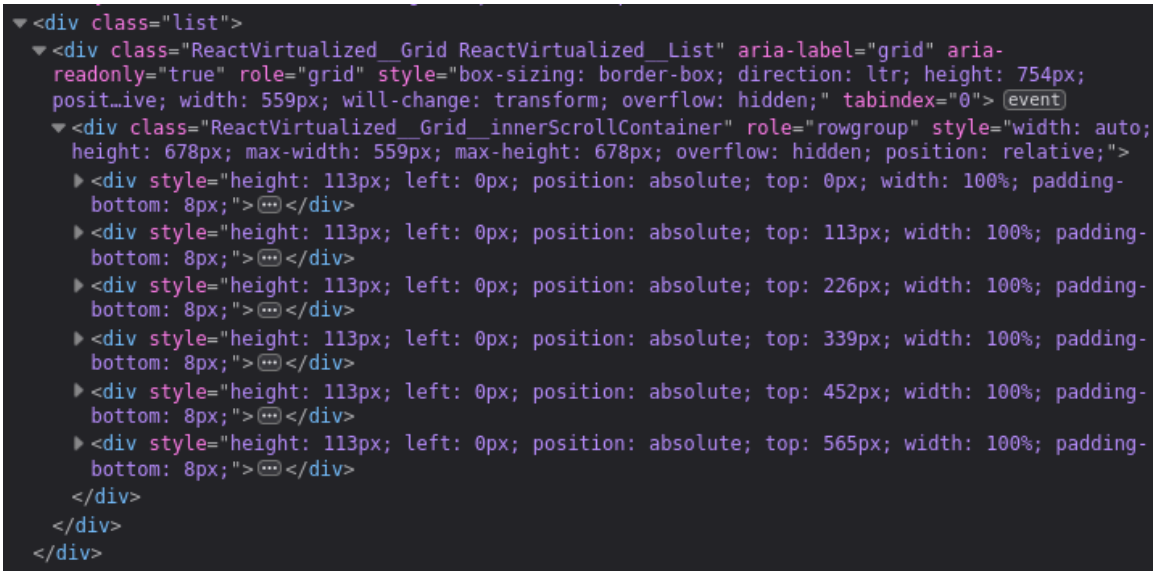

Figure 4.1: Only the segments that are within the user's view are rendered.

This approach is particularly critical given that the earlier version of the application attempted to render all segments upfront — potentially thousands in some cases. Rendering such a high number of segments simultaneously not only drastically increases memory usage but also causes sluggish performance and poor user experience. The revised logic circumvents this by skipping offscreen elements entirely, which aligns with classic virtual scrolling strategies seen in modern front-end development. For users, this means smoother navigation and a snappier interface, even when dealing with extremely long audio files.

This refactor achieves a more predictable rendering system. By prioritizing visible content and offloading invisible regions, the app ensures that performance remains consistent regardless of the size of the audio file.

4.2.2 Segment Text Overflow

In the legacy platform, adding new lines to a segment resulted in its contents overflowing and the segment height not adjusting to the changes (see figure 4.2). The added `updateListLayout` function ensures any changes to the segment contents trigger a layout update (see figure 4.3).

```
const updateListLayout = () => {
    if(!cellCache.current || !listRef.current)
        return;
    cellCache.current.clearAll();
    listRef.current.forceUpdateGrid();
}
```

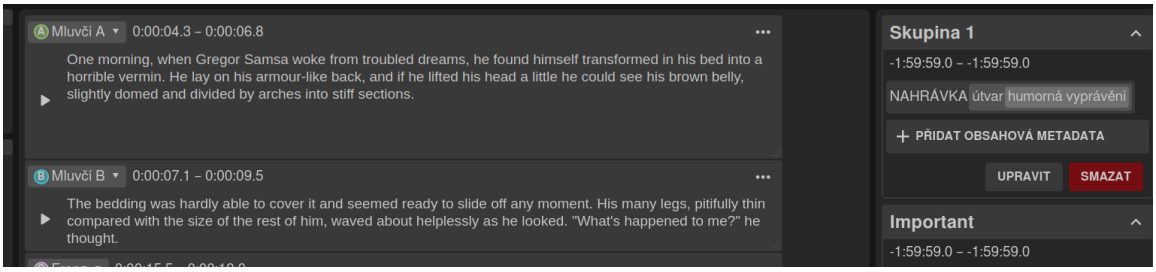

Figure 4.2: Segment height doesn't adjust to its contents.

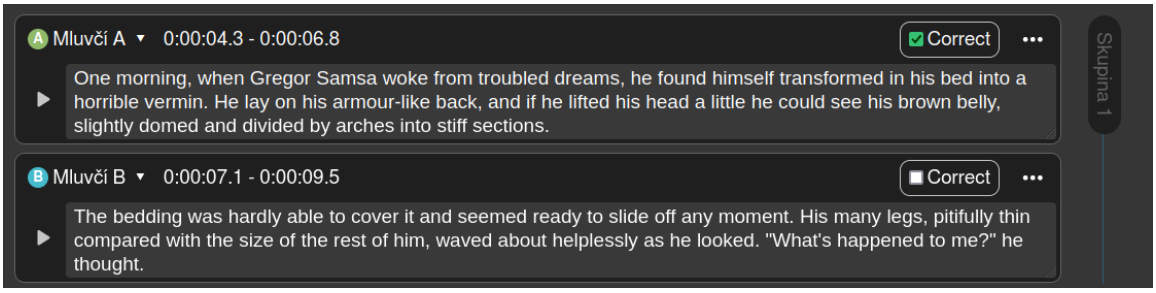

Figure 4.3: Text segments expanding/contracting based on their contents. Metadata height adjusts itself accordingly.

The newline characters added to the segment text were getting converted into spaces. This has been solved by changing the implementation of the `string2SegmentWords` function.

```
const tokens = text.trim().replaceAll("\n", "<br/>").split(" ")
```

This modification ensures that newline characters are preserved and rendered correctly in the browser. By replacing them with HTML line breaks before splitting the string, the interface can now reflect multi-line text inputs within segments as intended, preserving the structure and readability of the content.

4.2.3 Loading Animation

The `ClipLoader` component from the `react-spinners` library was selected to ensure that users receive immediate visual feedback during waveform or segment loading without overwhelming the interface. The spinner(see figure 4.4) is returned after all the hooks have been initialized.

Figure 4.4: Transcript loading animation.

4.2.4 Segment Layout

In the original implementation, the segment layout was handled using a CSS `Grid`. Each segment was wrapped in a React `Fragment` (`<>...</>`), which acts as an invisible wrapper and does not add any additional nodes to the DOM [4, 21]. When rendered, the resulting DOM structure appeared approximately as follows:

```
<div> <!-- This is the list container -->
  <div>...</div> <!-- Segment item without marker -->
  <div>...</div> <!-- Segment item without marker -->
  <div>...</div> <!-- Segment item with marker -->
  <div>...</div> <!-- Marker associated with the segment above.
                   Styled to appear to the right of the segment, not as separate list item
                       -->
  <div>...</div> <!-- Segment item without marker -->
</div>
```

React `Fragments` (`<>...</>`) are useful for grouping elements without adding extra HTML wrappers, such as `<div>` or `<span>`, helping to keep the DOM clean and minimal. In the current version of the layout, the use of a grid has been replaced by absolute positioning. Segment markers have been adjusted to the new positioning. Although it can introduce complexity, particularly in dynamic layouts involving elements like markers, absolute positioning has been selected as it allows precise control over element placement. When creating segments, the `requestAnimationFrame()` function ensures DOM updates and adjusts the viewport.

4.2.5 Autosave Setting

To prevent potential data loss, an automatic save feature has been implemented in the platform. By default, this autosave mechanism is triggered every 30 seconds, ensuring that recent changes to the transcript or metadata are periodically stored without requiring manual intervention.

Users are provided with the opportunity to manage this feature according to their preferences. Within the settings menu, a dedicated checkbox(see figure 4.5) allows users to enable or disable the autosave functionality. When disabled, changes must be saved manually, offering users full control over when data is written to the server.

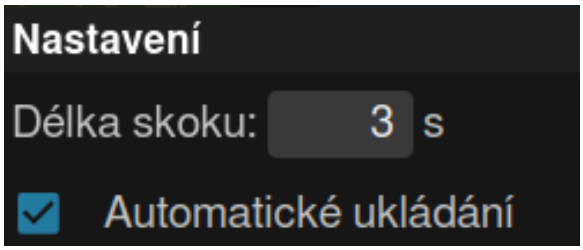

Figure 4.5: Automatic save setting.

In addition, a supplementary configuration option(see figure 4.6) has been introduced for users who wish to adjust the autosave behavior. If autosave is enabled, users can specify the interval at which changes should be saved. This interval is adjustable via a text menu, giving users the freedom to choose a save frequency that best suits their workflow.

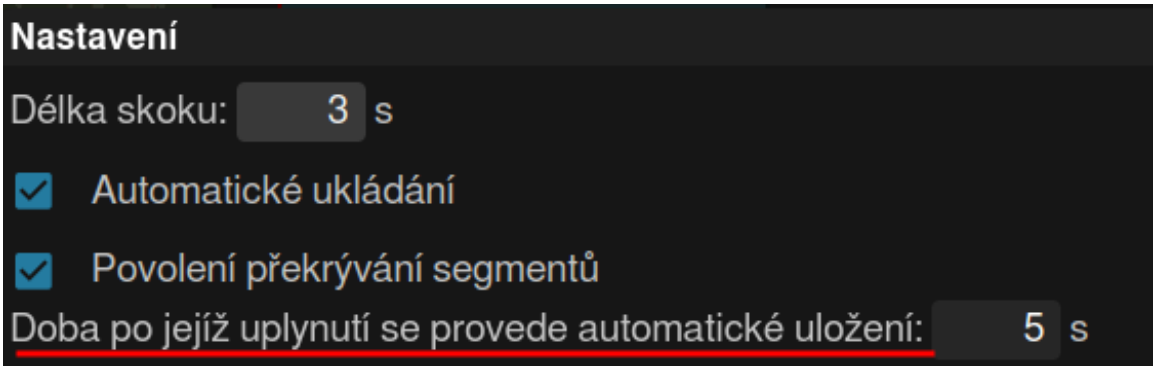

Figure 4.6: Automatic save time period modification setting.

4.2.6 Segment Group Deletion and Persistence

Group deletion has been implemented with persistence, ensuring that deleted groups remain removed across sessions by correctly writing to the Redux store. When attempting to delete segment groups, only the first group was successfully removed. This issue was caused by a variable shadowing bug in the implementation.

```
const idx = lookup.keys.findIndex(groupID => groupID === groupID)
```

This is a common pitfall in JavaScript and TypeScript, where a parameter name used in a nested scope unintentionally overrides a variable from an outer scope. In this case, both the callback parameter and the variable it was compared against shared the same name, resulting in a condition that was always true and thus returned the first match regardless of the intended logic.

The issue was resolved by renaming the inner parameter to a more descriptive and distinct name, avoiding any scope conflict. This type of bug is particularly troublesome because it fails silently—the code executes without any runtime errors but yields incorrect behavior, making it harder to detect and debug than more explicit failures.

In the previously deployed platform the segment group start and end timestamps didn't persist after refresh (see figure 4.7). `time2SegmentID` function has been reimplemented to allow for more precise segment matches so that the segment start and end don't fallback to „" after refresh resulting in incorrect segment group boundaries (see figure 4.8).

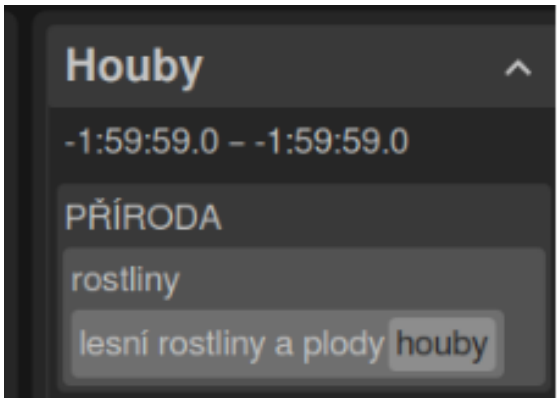

Figure 4.7: Segment group start and end being reset to „" after refresh.

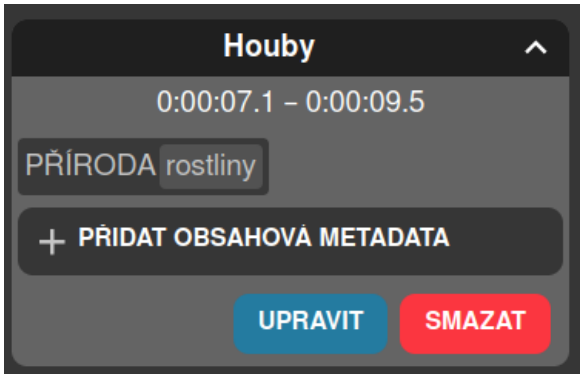

Figure 4.8: Segment group boundaries persist after refresh in the new platform.

4.2.7 Segment Mapping Updates

When adding a new line to a segment that already belongs to a group, the marker would previously incorrectly extend beyond the segment's boundaries instead of stopping at the appropriate position. This issue was resolved by rendering markers at the start of their respective groups and dynamically calculating their height based on the actual positions of segments in the DOM. Additionally, manual height calculations were removed to ensure the system remains independent of varying segment heights. Instead, the segment's `totalHeight` is now passed directly to the marker component for accurate rendering (see figure 4.9).

Figure 4.9: Marker height adjusts to the height of its segment members.

4.2.8 Selecting a New Speaker

To improve the annotation workflow, the implementation has been modified so that when a new segment is created, it automatically inherits the speaker label from the preceding segment(see figure 4.10) rather than defaulting to a preset speaker.

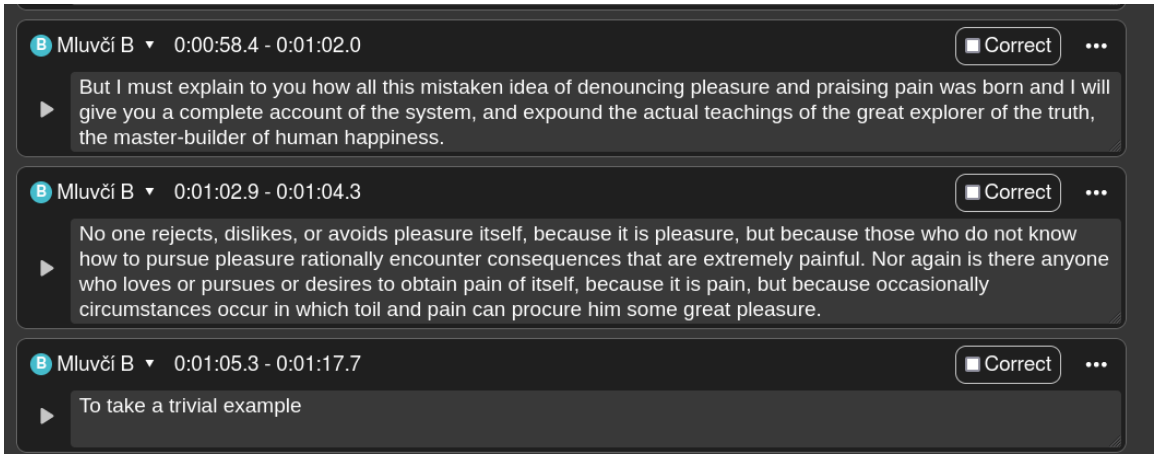

Figure 4.10: Newly created segments inherit the previous speaker.

This change reduces the amount of repetitive manual input required from annotators, particularly in cases where multiple consecutive segments belong to the same speaker, thereby streamlining the transcription process and minimizing potential labeling errors.

4.2.9 Notifications after save

A system of popup notifications has been implemented that are triggered whenever a saving event occurs. These notifications serve to inform the user in real time that their data is being saved, enhancing the overall user experience by providing immediate feedback and reassurance about the application's state (see figure 4.11 and 4.12).

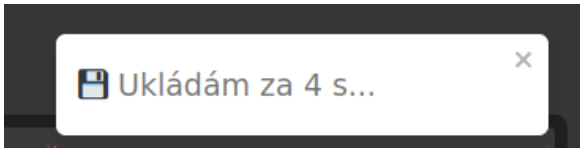

Figure 4.11: Countdown before saving the changes.

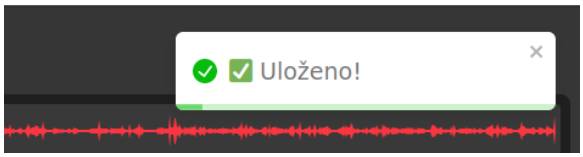

Figure 4.12: Successful save message.

4.2.10 Storing the last timestamp

The application utilizes `localStorage` [31, 23, 14, 26, 16] to retain the most recent playback or editing timestamp, ensuring that after a page refresh, the user is returned to the exact point where they left off. This enhances continuity and prevents disruption in the annotation workflow(see figure 4.13).

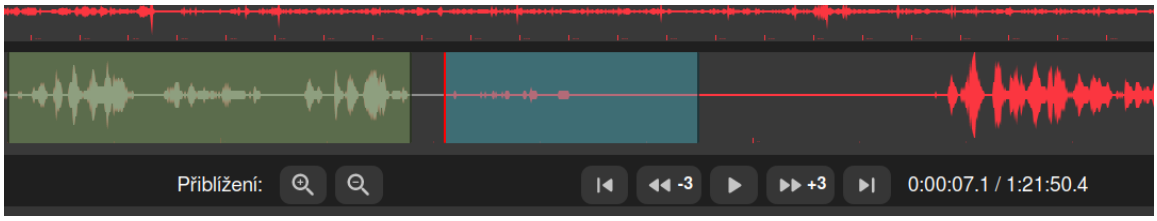

Figure 4.13: Waveform timestamp persists through user sessions.

4.2.11 Scrolling in the Waveform

Scrolling within the waveform component is now functional. The issue was resolved by intercepting scroll events before they propagate to the underlying DOM elements, allowing for more precise control over the waveform behavior.

The `Ctrl + Scroll` combination enables scrolling within the waveform (see figures 4.14 and 4.15).

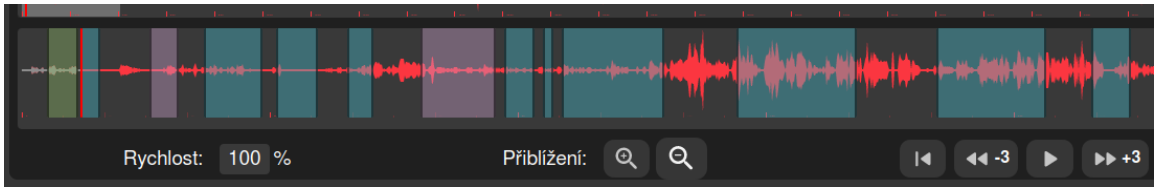

Figure 4.14: The waveform before scrolling

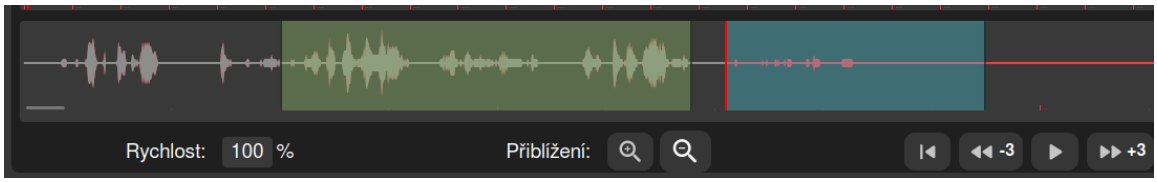

Figure 4.15: The waveform after scrolling

Another form of movement within the waveform has been introduced supporting the `Shift + Scroll` combination, offering horizontal navigation within the waveform.

4.2.12 Palettes

Palettes[1] and color schemes [2] have been used universally across the application. The application consistently utilizes palette color schemes throughout its interface.

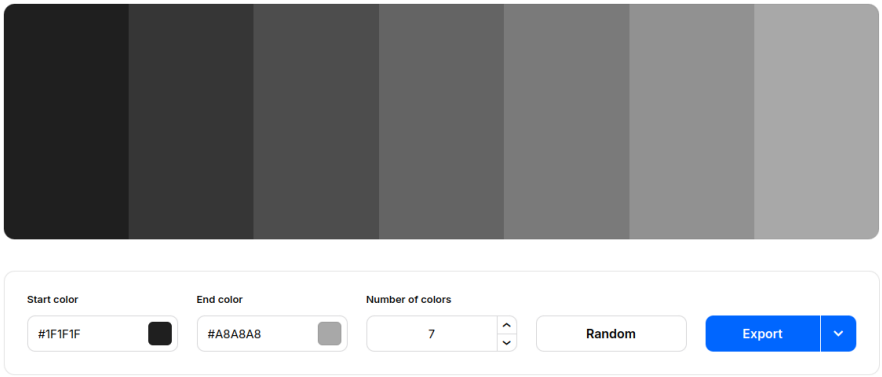

Figure 4.16: The palette used for **black and white** visual elements.

Figure 4.17: The palette used for **colored** visual elements.

[1] https://coolors.co/gradient-palette/1f1f1f-a8a8a8?number=7
[2] https://coolors.co/0a2463-fb3640-605f5e-247ba0-e2e2e2

This approach ensures a cohesive user experience, increasing usability across different components and sections of the platform. [8]

4.2.13 Segment rendering issues

It was noted that waveform regions to the left of the current timestamp did not get rendered correctly. This required a recalculation of the visible range. The issue stemmed from incomplete visible region calculation during the initial render and updates. Padding has been added to the visible range to ensure segments before/after the current time are included in the initial render. *Resize Observer* detects waveform container size changes and triggers a redraw whenever region width changes.

`loadVisibleRegions` Modification (Added Padding)

Without padding, the visible range calculation was too strict:

```
segment.start < visibleRangeEnd && segment.end > visibleRangeStart
```

This only rendered segments fully overlapping the exact visible viewport, missing segments near the edges.

Adding a **10% padding** to the visible range solved the issue:

```
const visiblePadding = duration * 0.1;
const visibleRange = {
    start: Math.max(0, visibleRangeStart - visiblePadding),
    end: Math.min(duration, visibleRangeStart + ... + visiblePadding)
};
```

The visible area was expanded to include segments just outside the immediate viewport. Segments before the current timestamp now fall into this padded range and get rendered.

Resize Observer

Waveform container size changes (e.g., window resize, UI adjustments) weren't triggering re-renders of segments. `clientWidth` calculation became stale. A resize observer has been added:

```
new ResizeObserver(() => {
    loadVisibleRegions();
    wavesurfer.current?.zoom(0); // Force redraw
});
```

This enabled to detect container size changes and:

- Recalculate visible regions using the **new container dimensions**.
- Force a redraw with `zoom(0)` to refresh the waveform canvas.

The combination of these two fixes ensures broader render coverage of segments near the current timestamp (before and after). Container resizes (which affect visible region calculations) became responsive and immediately refresh the rendered segments.

4.3 Introducing New Front End Functionality

4.3.1 Segment tags

The number of segment tags isn't known upfront. Yet the system needs to be able to render a variable number of segment tags (see figure 4.18).

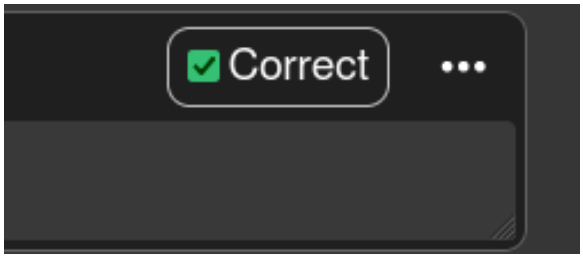

Figure 4.18: Transcript segment with a segment tag.

All tags have an identical interface and they're part of a `Grid`. After each interaction a new request is issued to the API to propagate the state. The clicked checkbox is stored in the Redux store. Higher count of segments will shrink to fit inside the grid space.

Implementation details

New fields have been added to `History` definition, as well as `dispatch` calls to `useUndoRedo`. `transcriptSlice` functionality has been extended, including new transcript fields and reducers. `toggleSegmentTag()` function has been introduced to switch the state of the checkbox itself. A new component `TagComponent` has been added to fetch the state whether the segment tag is checked.

4.3.2 Deleting Segments that Belong to Segment Groups

The deletion logic for individual segments and segment groups has been refined. When a segment that belongs to a segment group is deleted, the corresponding segment group is automatically adjusted or removed, depending on the scenario (refer to 4.19).

Possible Scenarios

- If the segment belongs to a group consisting of a single segment, the entire group is deleted in a cascading manner.
- If the group contains multiple segments, the group's boundaries (start and/or end) are updated to reflect the removed segment.

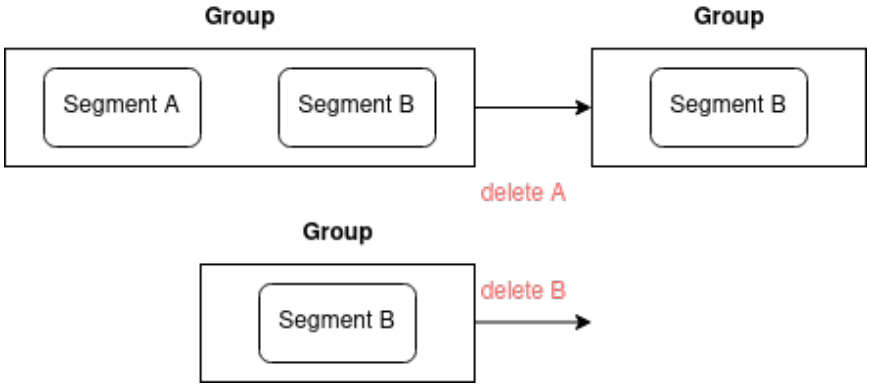

Figure 4.19: Group update scenarios after segment deletion.

For instance, if the start/end segment of a group is deleted, the group persists and adjusts accordingly. However, if both the start and end segments are removed, the group is considered invalid and is subsequently deleted.

4.3.3 Region Overlap

An additional setting that was introduced is the ability to enable or disable region overlap within the waveform interface (see figure 4.20).

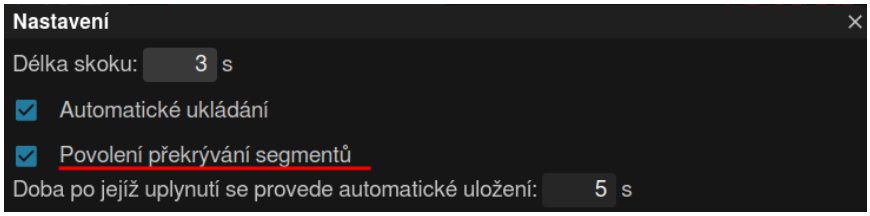

Figure 4.20: Checkbox to enable or disable region overlap.

By default, overlapping segments are disabled. Consider the case where there are two segments: `B(x3..x4)` and `A(x1..x2)`, where segment A lies to the left of segment B. If segment A is extended such that `x2 > x3`, its end time will be automatically adjusted—trimmed—so that `x2 = x3`, ensuring no overlap occurs(see figure 4.21).

A checkbox has been added to toggle this feature.

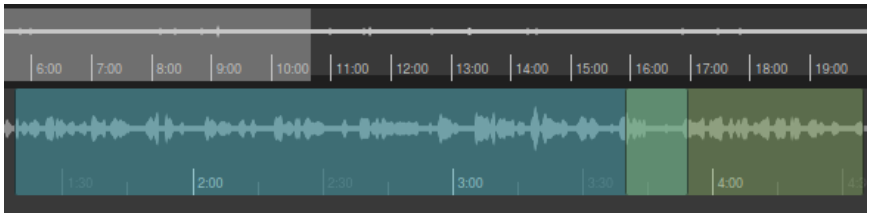

Figure 4.21: Example of overlapping waveform regions.

The programmatic behavior depends on the boolean overlap setting. An illustration of overlap functionality is shown at figure 4.22.

- **Overlap allowed:** The user can extend a segment freely, even into the bounds of another segment. The new segment will be recorded with its specified time range, and segments will remain sorted by their start times.
- **Overlap disallowed:** When a segment is extended and hits the boundary of an existing segment, it will automatically be trimmed to prevent overlap.

Figure 4.22: Overlap variability illustration.

4.3.4 Segment Movement Operations

Drag and resize operations have been introduced for waveform regions. Firstly they depend on the aforementioned setting that determines whether segment overlap is allowed. Resizing refers to manual movement of either of a region's boundaries. Performing a dragging operation results in both segment boundaries changing simultaneously.

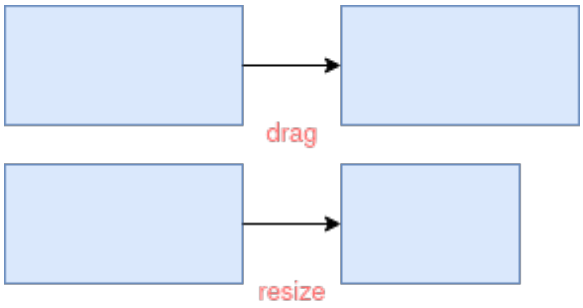

Figure 4.23: Drag and resize segment operations.

Synchronization of waveform and the segment transcript has been fixed. Each drag/resize operation gets propagated into the transcript. `handleResize` function handles the resize operations, while `handleSegmentMove` takes care of segment dragging.

Additionally, segment borders can be resized: moving segment borders in the waveform has been correctly synced with the Redux store, thus the transcript timestamps get updated accordingly.

4.3.5 Special Characters Customization

New functionality that allows users to directly customize special characters used in transcription has been implemented. This feature provides the flexibility to not only modify existing special characters but also add entirely new ones, including user-defined *tokens* [2].

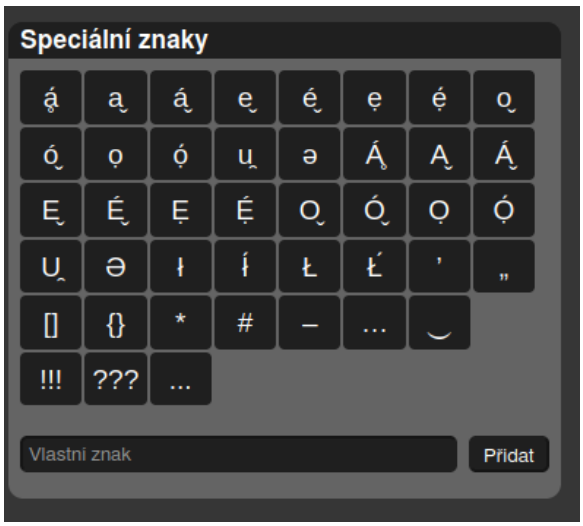

Figure 4.24: Special character list. New tokens can be created by the user.

Special characters in this context refer either to nonstandard symbols or *tokens*. Tokens are predefined sequences of characters that are treated as a single special character. Tokens can represent frequently used character combinations, words or expressions and are especially useful during annotation. When a user clicks on a token, several characters are inserted at once, significantly accelerating the transcription process by reducing repetitive typing.

4.4 Proxy Server for Real-Time Collaboration

4.4.1 Motivation for cooperative mode introduction

To address limitations in the previous platform and enable real-time collaborative editing [18, 5], a proxy server has been implemented using Django [3] that acts as an intermediary between the clients and the backend API (see figure 4.25). From the research of existing solutions described in section 2.2.6 it became clear that most annotation platforms do not provide cooperative writing thus adding the feature would make a difference. This server maintains the current state of the application and ensures synchronization of data across all connected user instances. It achieves this through a broadcasting mechanism that will be described in detail later. Additionally, collaborative editing tests that were performed are described in the chapter 5.

The need for such a solution arose from a critical issue in the earlier implementation: when multiple users opened and edited the same recording simultaneously, each instance would diverge independently, leading to unsynchronized and conflicting versions of the transcript. This behavior was undefined and often resulted in data loss and inconsistencies.

In the non-collaborative version of the application, if two instances of the application are open in different windows, changes to a the transcript in one(e.g. editing segment text) do not propagate to the other even after save and refresh operations.

In the updated architecture that functions as a separate application, clients no longer communicate directly with the backend API. Instead, each client connects to the Django server and provides the relevant `job_ID`. From that point onward, the Django server manages all communication. Any changes initiated by a client are first sent to the server, processed there, and then broadcast to all connected users. This ensures that every client always has the most recent version of the shared transcript, enabling collaborative writing.

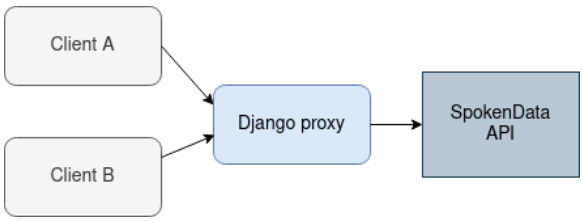

Figure 4.25: Data flow in the cooperative mode.

Additionally, leveraging a state management system like reducers allows for tracking local changes and emitting corresponding events from the front end side, which could then be mirrored to the server.

The shift in communication from *client-to-API* to *client-server-API* not only enables real-time cooperation but also significantly improves the performance and responsiveness of the application on the client side. By offloading logic and coordination to the server, the client becomes lighter and more efficient, resulting in smoother user interactions. Moving all API-related responsibilities to the Django server introduced significant architectural changes. However, since initial data formats remained the same, swapping the communication method didn't require a complete rewrite of the functionality.

Moreover, relocating key aspects of functionality from the frontend to the Django backend has provided other benefits:

- **Code Cleanliness and Maintainability:** Centralizing business logic on the server helps maintain a cleaner, more modular codebase.
- **Scalability and Extensibility:** Future features, such as role-based access, can be more easily integrated on the server side.
- **Execution Efficiency:** Centralized processing reduces redundancy and minimizes the computational load on each client.
- **Improved UX:** Clients benefit from faster load times and more fluid interactions due to reduced processing demands.

This architectural shift lays the foundation for a multi-user transcription platform, suitable for real-world collaborative use cases.

4.4.2 ASGI

The Asynchronous Server Gateway Interface (ASGI) [1] is a standardized interface specification that defines how web servers communicate with asynchronous Python applications

and frameworks. It was developed as a successor to the older WSGI (Web Server Gateway Interface), which was limited to synchronous request handling.

ASGI introduces support for asynchronous, event-driven programming, making it well-suited for handling modern web application requirements such as real-time communication, WebSockets, long-lived connections, and background task execution. Enabling asynchronous I/O allows for better performance and scalability, especially in real-time interactivity scenarios.

ASGI is now widely adopted in the Python ecosystem and forms the backbone of many modern frameworks, such as Django Channels, FastAPI, and Starlette. It acts as a flexible bridge between web servers (like Daphne) and asynchronous applications.

4.4.3 Websockets

WebSockets are a modern communication protocol that enables full-duplex, persistent connections between a client (typically a web browser) and a server. Unlike traditional HTTP, which is a request-response model, WebSockets allow for real-time, bidirectional data exchange over a single connection. Events and updates are transmitted as WebSocket messages, allowing for low latency. [11, 6]

This technology is especially useful for applications where low latency and live updates are crucial — such as collaborative editing tools.

The WebSocket protocol begins with an HTTP handshake, after which the connection is „upgraded“ to a WebSocket. Once established, both the client and the server can send messages to each other independently, without the overhead of repeatedly opening and closing connections(see figure 4.26).

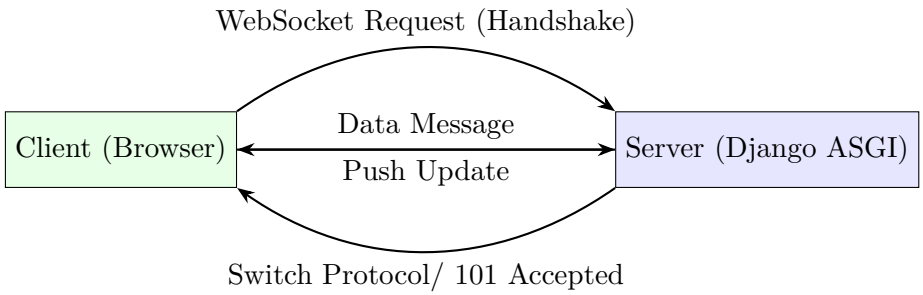

Figure 4.26: **WebSocket** Communication Flow Between Client and ASGI Server.

In Python-based backends, ASGI-compatible servers (such as `Daphne`) and frameworks like `Django Channels` are commonly used to implement WebSockets. This framework allows the server to manage multiple open connections simultaneously using asynchronous event loops, thus enabling high scalability and responsiveness.

The WebSocket API is natively supported in all modern browsers, making it an ideal choice for building real-time web applications. With WebSockets, developers can efficiently push changes from the server to all connected clients — a key feature in applications involving collaborative work and live data synchronization.

4.4.4 Daphne motivation and use

In a collaborative writing environment where multiple users edit and view content in real time, asynchronous, bidirectional communication becomes essential. Traditional WSGI-

based Django deployments are insufficient for this use case because they do not support asynchronous connections like WebSockets.

Daphne [7] is a production-grade HTTP/WebSocket protocol server designed for the asynchronous layer (ASGI) of Django. Its use is motivated by the following:

- **WebSocket Support:** Enables real-time, low-latency communication between users and the server—crucial for collaborative text editing.
- **ASGI Compatibility:** Daphne serves as the entry point for ASGI applications, allowing the Django project to leverage asynchronous views, background tasks, and message handling.
- **Concurrency Handling:** Unlike WSGI, Daphne can handle thousands of open WebSocket connections, making it ideal for situations where many users are connected to shared documents.
- **Pluggable and Scalable:** It integrates well with channel layers (e.g., Redis), enabling horizontal scalability and message broadcasting.

Daphne enables Django to function as a real-time collaboration backend, transforming it from a purely request-response framework into a live, interactive platform.

4.4.5 Client identification

Transitioning communication from the current API to the new server would mean that the client interface (GUI) becomes primarily responsible for sending and receiving updates through this server, rather than communicating directly with an external API. This opens the possibility for state centralization. When a client creates a new project (e.g., by providing a voice file URL), the server validates and stores it, and then distributes project data to any clients that request it, using WebSocket connections.

Clients are identified by their port numbers after connecting to a Websocket:

```
2025-03-17 21:26:20,333 INFO Total clients: 2
2025-03-17 21:26:20,343 DEBUG Sent WebSocket packet to client for ['127.0.0.1', 35692]
2025-03-17 21:26:20,343 DEBUG Sent WebSocket packet to client for ['127.0.0.1', 36218]
```

From a security perspective, authentication is handled via session tokens, which are passed during the initialization of the Websockets and are subsequently validated by the server using middleware such as Django Channels authentication layers. Once authenticated, a client's identity and permissions are embedded in the WebSocket scope for the duration of the session. Authorization mechanisms ensure that the clients can only access and modify projects they have permission for. Enforcing `wss://` (`WebSocket Secure`) connections ensures secure transport, utilising TLS to prevent man-in-the-middle attacks. Additionally, server-side logging and rate-limiting is used to detect suspicious activity and mitigate denial-of-service (DoS) risks.

4.4.6 Implementation outlines

The current architecture does not use a fully-featured database, instead it operates with a session-based model. When a user initiates a connection, they submit a request containing the URL of the transcription job they wish to edit. The client informs the server of its intention to edit a specific resource, prompting the server to create a **session** using the

resource's ID as the session key. The server then fetches the relevant data from the external API, stores it temporarily in memory, and responds to the client with the session's data.

Once the session is initialized, the client can begin editing the transcript. Each modification is transmitted back to the server, which updates the in-memory session state. When additional users request access to the same resource, the server checks for an existing session and, if found, attaches the new client to that session within a dedicated channel, delivering the current synchronized state. From this point on, any changes made by one client are sent to the server and broadcast to all connected clients, creating a real-time collaborative editing environment (see figure 4.27).

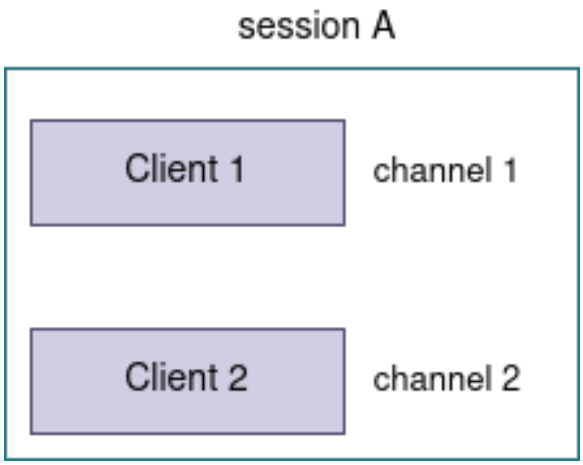

Figure 4.27: Session, channel and client hierarchy demonstration.

This mechanism builds on an internal **mapping of session IDs to their corresponding states**. Each session is represented as an in-memory object (such as a Python dictionary) that reflects the current state of the edited resource. When the server receives a request for a transcription with a specific ID, it either initializes the session by retrieving data from the backend API, or connects the client to an already existing session. For example, the internal session state might look like the following:

```
sessions["A"] = {
    "id": "A",
    "title": "Example_recording.WAV",
    "description": "",
    "category": "",
    "transcription": "",
    "status": "done",
    ...
}
```

Although a basic in-memory data structure such as a Python `dict` suffices for this initial implementation, more advanced options like an in-memory data store can be considered. **Redis** is a popular choice for this kind of session handling. Alternatively, the open-source Redis-compatible project **Valkey** can be used for storing and caching session objects. Regardless of the storage backend, the architecture remains simple and efficient: its primary responsibility is to maintain a live, shared session state accessible to multiple clients in real time.

4.4.7 Realization steps

The Django server holds the authoritative state and pushes updates to connected clients via WebSockets. Using Django `Channels` is a common solution to handle real-time communication and broadcast events whenever a client makes a change. In this model, whenever a user edits the transcript (or modifies metadata), the client sends a message via WebSocket to the server. The server then updates its main state and broadcasts that change to all connected clients so they can update their local views immediately.

The Django server will manage user channels to listen for changes. When a change occurs, a Django consumer can process the event and use a publish/subscribe mechanism to send a WebSocket message to all clients. On the client side, the application should subscribe to these events so that it can instantly synchronize the view. This architecture supports real-time collaborative editing and scales to multiple simultaneous users.

4.4.8 Folder structure

The Django server folder structure is organized according to Model-View-Controller architectural pattern, however mostly the term Model–Template–View (MTV) is used. [30, 24, 27] In this framework, the routes are defined in `urls.py` which maps URL patterns to *view functions*. These functions, written in `views.py`, act as controllers, handling incoming HTTP requests, implementing the logic, and sending appropriate responses. To fulfill such requests, a view often interacts with the model layer, defined in `models.py`. Models are Python classes that define the structure and relationships of the underlying SQL database using Django's Object-Relational Mapping (ORM).

Presentation logic is handled by Django's template system, which allows embedding Python expressions into HTML to dynamically generate content. Additional components extend the MVC pattern to support real-time communication. For instance, `consumers.py` uses methods such as `receive()`, `get_client()`, `broadcast()` that handle client-server communication to defines WebSocket consumers. `messages.py` defines structured message schemas (e.g. `LoadJobMessage`) for consistent data exchange. `data_store.py` implements the application logic and state management via classes like `JobManager, JobChannel, JobClient`. `spokendata_api.py` is used for direct communication with the API from the server.

```
collab_transcript
└── manage.py
    ├── __init__.py
    ├── asgi.py
    ├── consumers.py
    ├── data_store.py
    ├── messages.py
    ├── routing.py
    ├── settings.py
    ├── spokendata_api.py
    ├── urls.py
    └── wsgi.py
```

Figure 4.28: Tree structure of the cooperative writing folder.

Since the use case is multi-user real-time collaboration, this separation reduces cognitive load: developer knows exactly where to look for specific features.

4.4.9 WebSocket integration into TypeScript

When integrating WebSockets into a TypeScript-based frontend application, a common pattern involves defining a socket interface that abstracts connection logic, message serialization, and event management. This approach ensures type safety and improves code maintainability by clearly defining message types and event structures.

The following code snippet outlines the used process of connecting to a WebSocket server, handling its readiness, registering event listeners, and responding to incoming messages.

```
if(socket.isDisconnected()) {
    socket.onReady = (e) => {
        socket.send(new LoadJobMessage(JOB_ID))
        const handleUpdateEntities = (e: Event) => {
            const customEvent = e as CustomEvent<{ entities: any; }>;
            socket.send(new SaveTranscriptMessage(JOB_ID, customEvent.detail.entities, {}))
        }
        document.addEventListener('update-segment-entities', handleUpdateEntities)
        socket.onMessage = (e) => {
            const message = BaseMessage.fromJson(e.data)
            switch(message.messageType) {
                 case MessageType.SaveTranscript:
                    if (JOB_ID) socket.send(new LoadJobMessage(JOB_ID));
                    break
            }
        }
    }
    socket.connect("ws://localhost:8000/ws/testos/")
}
```

This implementation begins by verifying whether the socket is currently disconnected. If so, the connection lifecycle is initiated. Once the socket is ready, an initial message `LoadJobMessage` is sent to the server to retrieve the current state associated with a particular job identifier (`JOB_ID`).

Subsequently, a custom browser event named `update-segment-entities` registers a listener. This listener intercepts emitted events containing modified segment data (referred to as `entities`) and transmits them to the backend via a `SaveTranscriptMessage`. This pattern decouples UI interaction from network logic, promoting a reactive architecture.

Incoming messages from the server are parsed through a generic deserialization function `BaseMessage.fromJson()`. Depending on the message type received, appropriate actions are taken. For instance, when a `SaveTranscript` response is acknowledged, the client triggers a reload of the job state by sending another `LoadJobMessage`.

Finally, the socket is instructed to connect to the WebSocket endpoint located at `ws://localhost:8000/ws/testos/`. This completes the setup for real-time bidirectional communication, allowing the frontend to both react to and propagate changes as they occur.

This pattern forms the basis for integrating WebSockets into a TypeScript frontend and is further extended to support error handling, reconnection logic, authentication, and message queuing.

4.4.10 Delta-Like Segment Updates

In collaborative editing, a *delta* usually refers to the minimal difference between document versions—capturing what changed, where, and how. While traditional delta systems (e.g., Google Docs) compute precise insert/delete operations, the approach in this thesis simplifies this idea.

Segment-Level Updates

Rather than sending the entire document or full delta sequences, we update only the specific segment that changed. When a user edits a segment:

- The corresponding segment component detects and computes the change.
- It sends just the modified segment (not the whole transcript) to the server.
- This avoids unnecessary payload and preserves efficiency.

This approach isn't brute-force, there's no repeated full-state upload. Only the active unit (e.g., a sentence or timestamped region) is transmitted.

Server-Side Handling

On the server:

- Each segment update is received individually.
- The backend patches the updates into the job's current state.
- If applicable, the server broadcasts this change to other clients working on the same job.

This pattern provides many of the benefits of traditional delta approaches.

4.4.11 WebSocket Scope and Client Sessions

WebSocket sockets should be global and shared across the application, not scoped only to individual transcript segments or lower level components. This is critical for real-time collaboration.

Core Logic Flow

A centralized handler is responsible for sending messages, receiving messages via `onmessage`, parsing custom JSON messages (type, payload, etc.). After a user requests a `job_id`, the server responds with associated job data. The server then computes diffs (deltas) between edits and broadcasts them to all connected clients in real-time. `consumers.py` is used to propagate changes across sessions.

Session Management and State

The application maintains a persistent server-side state that includes a list of connected users, active job sessions, and tracked WebSocket sessions identified via IDs. To track users effectively, user records are added to the WebSocket consumer's state, and Django Channels

middleware is used to inject additional context into the `scope`. The scope object itself serves as a useful storage for connection data, and middleware can enrich it with user and session-specific context. Regarding job assignment logic, when a client requests a job, the server assigns one accordingly. Once the client begins making changes, the server—already holding the updated state—is able to efficiently broadcast those changes to other clients.

Client-Server Communication

Active WebSocket connections are managed through Django Channels. When a user connects to the server, the `JobManager` assigns them to the appropriate job. Communication between the client and server is handled by a WebSocket `Consumer`, which is responsible for managing interactions related to a specific model, such as a transcript. The consumer both sends and receives messages using `consumer.send()`. Additionally, the system must account for edge cases, such as unexpected client disconnections, to remain operational.

Job Channel Architecture

When a client requests a job that does not yet exist, the system dynamically creates a channel for the job and loads the associated data. Upon connection, the server transmits the relevant job data back to the client using `consumer.send()`. Because communication and data handling are asynchronous, all such operations are performed using async methods and must follow the `await` syntax to ensure correct execution flow.

4.4.12 Client-Server Interaction Flow

The typical WebSocket lifecycle begins when the client sends a `loadJob` request containing a job ID. The server first checks its local cache or queries an external API to determine whether the requested job exists. If it does not, the server creates a new job instance, assigns the client to it, and sends the corresponding job data—formatted as JSON—back to the client.

Clients should not fetch job data directly from the API. Instead, the server functions as the central intermediary: it manages all communication with external APIs and ensures coordination between multiple clients connected to and working on the same job.

4.4.13 Data Store System Overview

The data store system is designed to facilitate collaborative real-time editing of audio recordings over WebSocket connections. At the core of this system are three main components: `JobClient`, `JobChannel`, and `JobManager`. Together, these classes manage client connections, job assignments, and channel grouping for efficient and organized communication.

The `JobClient` class represents a client connected via Websockets. The user is linked to a channel corresponding to the job.

```
class JobClient:
    def __init__(self, consumer: AsyncWebsocketConsumer):
        self.id: UUID = uuid4()
        self.consumer: AsyncWebsocketConsumer
        self.channel: JobChannel = None
```

The `JobChannel` channel represents a group of clients simultaneously editing the same recording. For each job, the client list is managed.

```
class JobChannel:
    def __init__(self, job_id):
        self.job_id: str = job_id
        self.clients: List[JobClient] = []
```

The`JobManager` class acts a central orchestrator for WebSocket clients. It manages all connections and job associations.

```
class JobManager:
    def __init__(self):
        self.channels: List[JobChannel] = []
        self.unassigned_clients: JobChannel
        self.consumer_clients: Dict[AsyncWebsocketConsumer, JobClient]
```

Hierarchical Structure

The diagram below illustrates the hierarchical relationship between the core components of the system:

```
DataStore
 JobManager
     channels: List[JobChannel]
        JobChannel
            job_id: str
            clients: List[JobClient]
     unassigned_clients: JobChannel (special)
     consumer_clients: Dict[AsyncWebsocketConsumer → JobClient]
```

This structure ensures that:

- Each `JobClient` can be quickly looked up via its WebSocket connection.
- Clients can be dynamically moved between channels (jobs).
- Group-level operations (like broadcasting messages) can be performed per `JobChannel`.

4.4.14 Client Lifecycle

When a client connects to the server via WebSocket, it is added to the `unassigned_clients` list. Afterwards, it is registered in the main `consumer_clients` registry for active tracking and communication.

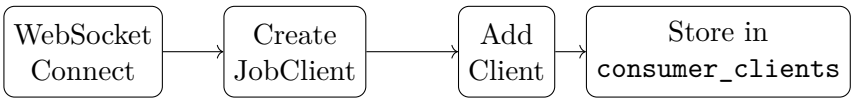

Figure 4.29: Client connection and registration flow.

Upon receiving a job change request from a client, the system determines whether a corresponding communication channel already exists. If so, it fetches it; otherwise, a new

one is created. The client is then reassigned to this job-specific channel, and internal tracking structures are updated accordingly.

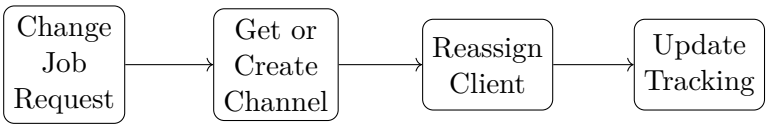

Figure 4.30: Job reassignment and client migration flow.

If a WebSocket connection is terminated (either intentionally or due to an error), the associated client is cleanly removed from both its current job channel and from the `consumer_clients` registry to maintain consistency and avoid stale references.

Channels are instantiated dynamically and only when needed. The architecture supports bidirectional tracking between clients and their assigned channels, enabling efficient communication. Access to a specific client is performed in constant time $\mathcal{O}(1)$, while locating all clients within a job-specific channel is a linear operation $\mathcal{O}(n)$, depending on the number of clients.

The following listing demonstrates the server-side flow when a client initiates a job load request. The server fetches the necessary job and transcript data from the API, returns the data to the requesting client, and broadcasts the update to all other clients in the same job channel.

```
Client A --> Server: LoadJobMessage(job_id)
Server --> API: GET /job/{id}, /transcript/{id}
API --> Server: job_data, transcript_data
Server --> Client A: job_data + transcript_data
Server --> Client B: broadcast(job_data + transcript_data)
```

To aid debugging and monitoring, all WebSocket communications initiated by the server are logged in the backend output. A sample log excerpt is shown below:

```
2025-03-17 21:26:20,333 INFO Total clients: 2
2025-03-17 21:26:20,343 DEBUG Sent WebSocket packet to client for ['127.0.0.1', 35692]
2025-03-17 21:26:20,343 DEBUG Sent WebSocket packet to client for ['127.0.0.1', 36218]
```

4.5 Conclusion

This chapter introduced the necessary theory to better explain the development process and laid out the applied changes. To build a collaborative editing system, WebSocket scoping, session tracking, and delta-based updates have been implemented. Using Django Channels with middleware and job assignment logic enabled real-time updates while maintaining a separation of concerns and a scalable architecture.

Chapter 5

Testing and Evaluation

By systematically evaluating the applied enhancements, the project strives to deliver a solution that is in line with modern standards and user expectations. This chapter describes the structure of the environment used to create the application as well as the details of the testing process.

5.1 Environment Structure

This section provides the context for the languages, frameworks and tools that were chosen for the solution implementation, as well as details about the project environment structure.

5.1.1 TypeScript

TypeScript and React have been selected for the project. TypeScript, being a statically-typed superset of JavaScript, introduces a layer of static typing that significantly enhances code maintainability. This type checking at compile time helps catch potential errors early in the development process, reducing the likelihood of runtime issues and fostering a more reliable codebase. Additionally, TypeScript lets users define interfaces and utilize advanced features, which improves general code quality. Furthermore, TypeScript's compatibility with modern JavaScript and its support for ECMAScript features ensure stable integration with existing JavaScript libraries and frameworks, allowing to leverage plenty of web development tools. [25, 28, 29]

React was chosen for this project due to its declarative syntax, component-based architecture, and efficient Virtual DOM. React's one-way data binding ensures predictable state management. React's compatibility with React Native allows for code reuse in the application, reducing development efforts.

5.1.2 Version Control System

`Github` has been used as the version control system. The Github repository where this project is developed has been divided into branches corresponding to the developed functionality. The app diverged at the introduction of cooperative writing functionality. Before the change, the application only consisted of TypeScript source files. Cooperative writing functionality involved creating a set of Python files implementing the Django server functionality described in chapter 4 that are located in the `coop_writing` branch. The non-cooperative functionality remained in the `standard` branch and `main` is the most stable

branch. Conventional commits[1] styling has been applied to standardize commit messages (see figure 5.1).

Figure 5.1: Standardized Github commits help facilitate new feature addition tracking.

Code is being periodically refactorized with the purpose of improving its quality. See figure 5.2 for the final version of the platform.

5.1.3 Project structure

At the highest level the project structure includes the `components`, `features`, `redux`, `style`, `types`, `utils` folders. Project components are organized in a tree structure.

Folders `components`, `style`, `types`, and `utils` contain files implementing app-wide functionality. The `redux` folder contains the Redux and the store setup. The `features` folder contains subfolders implementing app features. There are four feature subfolders, `grouping`, `player`, `transcript`, and `workspace`. Apart from these folders, there are also several files. `App.tsx` and `main.tsx` are standard React files serving as entry points for the application. `app.config.ts` stores an API key that is loaded when running the system locally. The config file is in `.gitignore` to prevent accidental key leaks. Each component is implemented in a separate file, which makes the application architecture modular.

5.1.4 Yarn: Dependency Management and Build Tooling

This project uses `Yarn` for managing JavaScript dependencies, scripts, and build processes. Compared to alternatives like `npm`, Yarn offers faster installations, deterministic dependency resolution through lockfiles, and better handling of workspaces, making it suitable for front-end systems.

[1] https://www.conventionalcommits.org/en/v1.0.0/

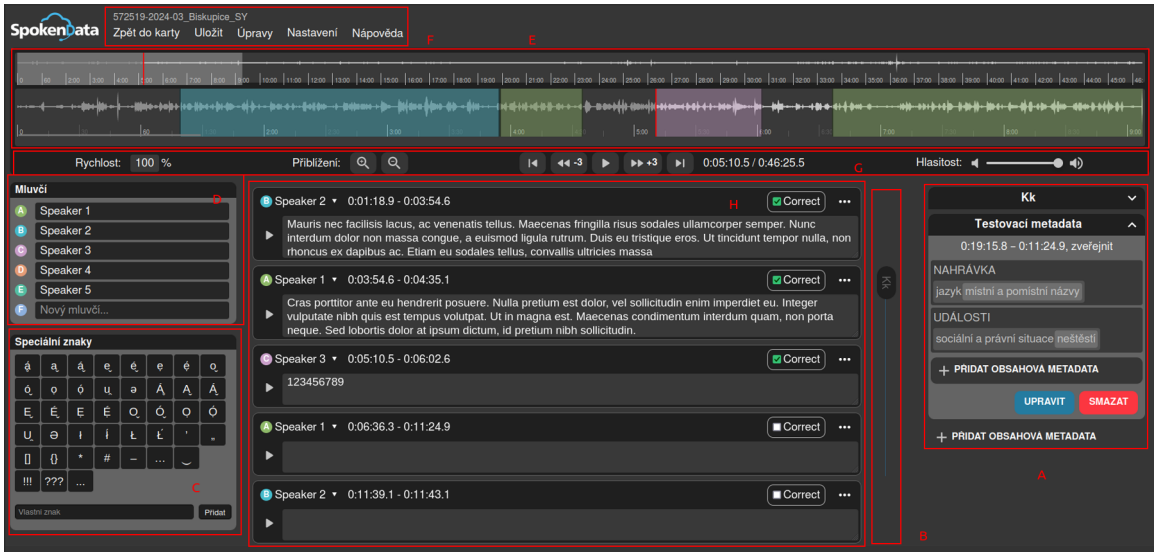

Figure 5.2: Final version of the application divided into sections. `A` is the segment metadata group list and group creation/modification form. `B` is the metadata marker highlighting groups of segments belonging to certain groups. `C` is the field containing special characters, `D` is the speaker menu, `E` is the waveform, `F` the control panel, `G` is the segment list and `H` is the segment list.

5.2 Priorities and Development Progress

The main development priority were segment operations. An improved functional version of the app has been provided in January 2025 that already had comparative improvements to the previous version of the platform. A very detailed feedback(see A) has been received afterwards that helped direct the development process.

5.3 Preliminary User Feedback on Segment Coloring

To explore the potential usefulness and reception of segment coloring within the interface, a few users have been consulted and shown a mock-up of the proposed feature (see Figure 5.3). The users have already been familiarized with the application and tried it out. The feature in question involved visually differentiating audio segments using colored overlays or stripes, in order to indicate the speaker identity of the segment. The question posed to the users was:

> Do you see this feature as useful, would it make your interaction with the platform easier?

Below is a summary of the user feedback gathered:

Respondent 1 (R1): The user expressed that the current layout already differentiates enough visibly between segment speakers and that adding more colored visual elements would clutter the layout and distract their attention.

Respondent 2 (R2): The user proposed to proceed with adding the feature to free up the space currently taken by the speaker selection button.

Respondent 3 (R3): The user argued that the feature wouldn't be useful as the overlay would only take a small portion of the segment and woudln't be as easily noticeable as the current selection button. Instead, the user suggested to make all segments colored according to their speaker.

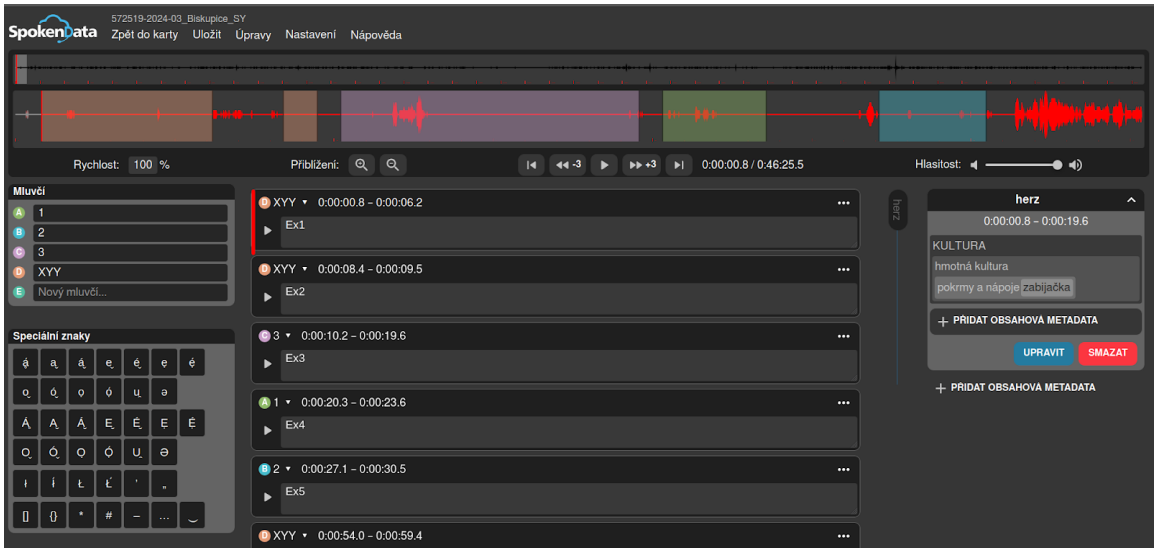

Figure 5.3: Mockup of potential segment coloring variations.

This preliminary feedback reflects a range of perspectives, from skepticism about visual clutter to enthusiasm for speaker-aware color coding. Since the majority of users didn't consider the feature necessary, it wasn't prioritized at this stage and could be added as a customizable option in the future. However, color coding entire segments according to their speakers was agreed to by all users, hence that can be implemented instead.

5.4 Feedback for the Final Version of the Application

In the final stages of implementing the editor, multiple feedback gathering sessions were organized to validate the user-friendliness of the developed layout.

The research techniques discussed in section 2.1.3 before have been used here. Specifically, observations, interviews and focus groups.

5.4.1 Testing process

To assess the usability and performance of the final implementation, a comparative evaluation has been conducted involving both quantitative benchmarks and structured user testing. Five users with varied experience in audio annotation were recruited. Each participant tested four systems:

1. The final version of the web-based annotation tool
2. The earlier implementation by Dugovič [17]
3. ELAN

4. Audacity

Each user interacted with all tools using approximately hour-long audio recordings. They were instructed to click at four distinct locations in the waveform and annotate a unique five-minute segment of the recording. The annotated segments were non-overlapping to avoid redundant effort and reduce bias from task familiarity. User feedback, both quantitative and qualitative, revealed clear advantages in favor of the current version.

Current Version vs. Dugovič's Implementation

Table 5.1: Quantitative Comparison: Dugovič vs. Current Version

Metric	**Dugovič (v1)**	**Final Version**
Load Time (avg)	17.9s	1.5s
Annotation Start Time	21s	2s
Crash Incidence	1/5 users	0/5 users

Selected User Feedback:

R1: The new version is much faster at loading transcript segments. The interaction is more predictable.

R2: Previously, scrolling and zooming caused noticeable lag and occasional crashes with longer recordings. Using the new platform made editing operations easier.

Quantitative Result: Average time to start annotating dropped from **21s** to **2s**.

Current Version vs. ELAN

Table 5.2: Quantitative Comparison: ELAN vs. Final Version

Metric	**ELAN**	**Final Version**
Ease-of-Use (1–10)	4.0	8.2
Team Collaboration Support	Manual Sync	Built-in
Setup Requirements	Java Runtime	Web-only

Selected User Feedback:

R2: The user appreciated not needing to install Java. Being able to run the annotator in a browser is beneficial.

R3: Elan does not appear intuitive for non-linguist users.

R5: Great for focus. Perhaps the new platform has a slightly better workflow.

Qualitative Result: User-reported ease-of-use improved from a median score of **4.0/10** to **8.2/10**.

Current Version vs. Audacity

Table 5.3: Quantitative Comparison: Audacity vs. Final Version

Metric	Audacity	Final Version
Average 5-min Annotation Time	41.4 min	32.6 min
Task Satisfaction (1–10)	6.2	8.7
Undo Safety	Medium (destructive)	High (non-destructive)

Selected User Feedback:

> **R1:** Audacity is more appropriate for cutting clips, but not for labeling them. The new platform is better suited for annotation purposes.
>
> **R3:** Audacity's irreversible undo mechanism isn't intuitive. The developed platform doesn't have that disadvantage.
>
> **Quantitative Result:** Users completed their 5-minute annotation **30% faster** in the current implementation compared to Audacity. Task satisfaction (based on a 1–10 scale) increased from **6.2** to **8.7**.

Conclusion

The final version of the annotation application outperformed both the prior implementation and established alternatives across all major metrics. Users consistently reported faster interaction and loading times, superior usability for longer audio content, greater confidence in undo/redo operations. These results demonstrate that the redesigned tool is not only technically more efficient but also offers a significantly improved user experience for the determined use case.

5.5 Load Performance Evaluation

This section provides a comparative analysis of page loading performance between the legacy transcription platform and the newly developed one. The goal of the evaluation is to assess how architectural and implementation changes—particularly around data loading and frontend rendering—impact user-perceived latency and network efficiency.

As is shown in the table (see 5.5), the new platform on average has higher loading performance than the legacy one. Overall time and segment load time have increased dramatically. DOM content rendering became constant thanks to lazy loading of segments.

Metric	100 Segments		200 Segments	
	Legacy	Current	Legacy	Current
Requests	162	162	161	162
Data Transferred (raw)	3.92 MB	8.02 MB	3.93 MB	6.52 MB
Data Transferred (effective)	2.73 MB	2.97 MB	2.73 MB	129.10 kB
Finish Time	3.15 s	2.19 s	2.87 s	2.00 s
DOMContentLoaded	489 ms	22 ms	759 ms	22 ms
Load Time	2.23 s	1.02 s	2.33 s	1.06 s

Table 5.4: Comparison of page performance metrics for the legacy and current platforms on 100 and 200 segment recordings.

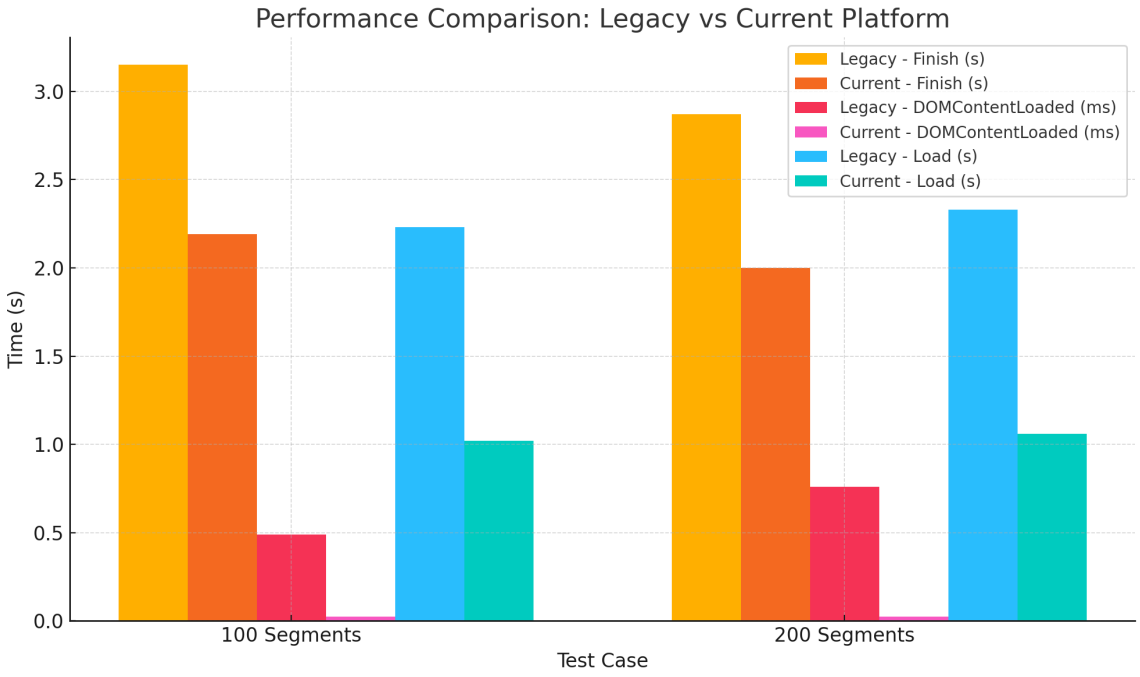

Figure 5.4: The new platforms presents a substantial improvement of loading performance.

Load time improvement averaged at **55%** (see figure 5.4) and DOM content load speed increased dramatically, approaching constant time (**95%** improvement and above for higher segment counts).

5.6 Academy of Sciences Testing

After the final version of the application had been presented, performance tests were carried out by linguists at the Academy of Sciences. All users are annotation professionals who, except one, have had experience with similar platforms in the past. The users were given the task to annotate short segment of audio across 4 recordings. For each recording, a segment of approximately 4 minutes was selected and annotated using their previous method and the new platform developed as part of this thesis.

The results have been the following:

Table 5.5: Quantitative Comparison: Microsoft Word + Audacity vs. Final platform

Metric	**Word + Audacity**	**Final platform**
Recording #1	1h25m	58m49s
Recording #2	45m45s	38m50s
Recording #3	53m27s	45m39s
Recording #4	40m21s	1h25m

The tests quantitatively showed the improved performance of the new platform in the order of **31%**, **15.2%** and **15%** accordingly (see figure 5.5), which averages around **20.4%**. The developed platform has made the work of the annotators more effective and improved their workflow.

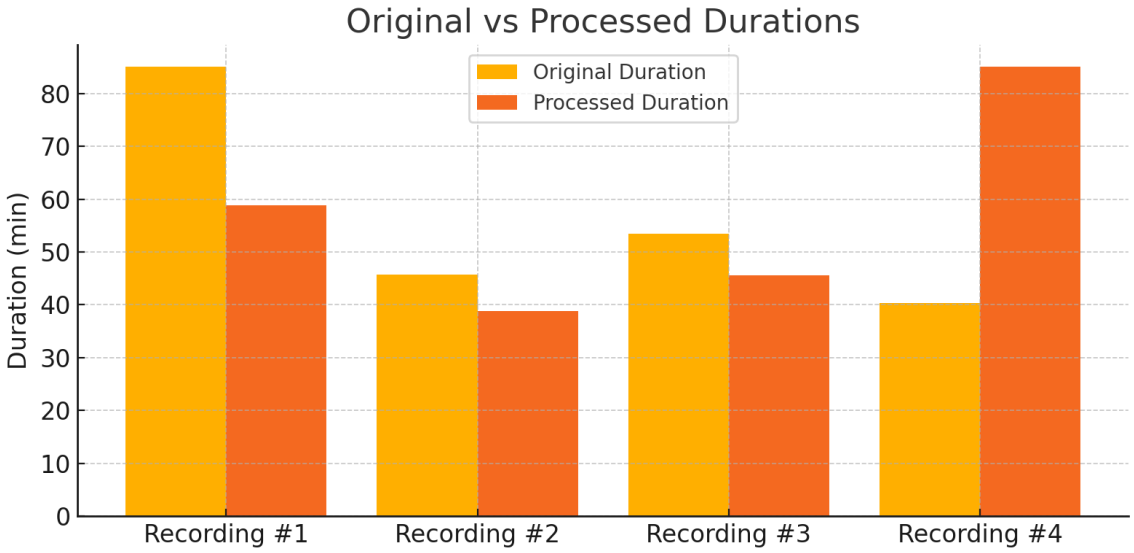

Figure 5.5: Quantitative improvement in the annotation time thanks to the current platform.

Since for the only non-experienced annotator the required time to complete their tasks actually increased, it's important to address non-experienced users. Therefore, a demonstrational video has been recorded to showcase the usage of the platform.

5.7 Collaborative Editing Testing

Tests were conducted to evaluate the performance of the cooperative version of the application. Three instances of the application were launched, with each of the three users assigned to annotate 20 transcript segments located at the 5th, 10th, and 15th minute marks, respectively. To simulate real-world collaboration, users were tasked with annotating different portions of the recording simultaneously.

In a baseline scenario, annotating 60 segments sequentially took approximately 7 minutes and 5 seconds. However, when the task was divided among 3 users working in parallel,

the total annotation time dropped to just 2 minutes and 45 seconds — a reduction of about **60%**.

5.8 Potential Future Improvements

The following is a prioritized list of potential future enhancements aimed at improving usability, scalability, and flexibility of the system:

- **Word-level tagging:** Inspired by systems like ATCO, allowing finer granularity in annotations.
- **Focused annotation mode:** Streamlined UI for annotating only short, relevant segments of audio.
- **Interface configuration support:** Allow users to modify the interface and behavior based on their use case or preferences.
- **Audio-optional mode:** Enable usage of the system even when no audio is available or necessary.
- **Stereo audio support:** Enhance playback and visualization for dual-channel recordings.
- **Multi-audio file support:** Allow synchronized playback of multiple files, enabling channel selection or composite views (e.g., multi-speaker setups).
- **Anonymization tools:** Enable zeroing or redacting sensitive segments of audio to support privacy and compliance needs.
- **Offline-capable backend:** Storing the required project libraries locally would remove the need to rely on external resources or libraries.

Conclusion

The implementation phase of this project has demonstrated the benefits of employing a robust development stack and adhering to best practices of software engineering. The choice of TypeScript and React has provided a strong foundation for building a reliable, scalable, and maintainable application. TypeScript's static typing and enhanced tooling have minimized potential runtime errors, while React's declarative syntax and component-based architecture have streamlined the development process and facilitated efficient state management. Furthermore, organizing project components hierarchically has ensured clarity in data flow, promoting modularity and ease of maintenance.

Version control through GitHub, coupled with conventional commit styling, has ensured a systematic and collaborative approach to development. Periodic code refactoring has enhanced code quality, addressing inefficiencies and incorporating feedback from testing. These strategic decisions have laid a solid groundwork for advancing the application while maintaining flexibility to accommodate future enhancements.

Chapter 6

Conclusion

The primary goal of this thesis was to further develop and enhance an existing audio-to-text transcription platform, originally implemented by Jakub Dugovič as his Bachelor's thesis in 2024. This work, conducted within the framework of the JARIN project at the Czech Academy of Sciences, aimed to optimize user interaction with audio recordings and elevate the overall user experience (UX), particularly in the context of large-scale linguistic annotation tasks.

Rather than building the system from scratch, the work involved evaluating, and systematically improving the inherited platform. An early phase of the project involved usability testing and a critical review of the current implementation, alongside comparative analysis of similar applications and relevant UX/UI design literature. These activities laid the foundation for a structured development process aligned with real user needs and research workflows.

The suggested literary sources and other sources that I have come cross during research helped to gain a more profound understanding of the UX/UI topic, as well as identify key principles influencing whether an interface might be viewed positively from the UX/UI point of view.

Since January, the updated platform has been deployed at the Czech Academy of Sciences, where it has already supported the annotation of multiple Czech audio recordings. It now serves as a vital tool for linguistic research, with specific effort made to preserve and document regional Czech dialects. The improvements implemented throughout the thesis work have significantly streamlined the annotation workflow and enhanced the application's practical value.

The development phase focused on building the application while incorporating continuous testing at both the unit and system levels. Testing was systematically carried out at the Institute of the Czech Language at the Czech Academy of Sciences, serving as a validation process to confirm the platform's accuracy, effectiveness, and value for its intended purpose. This ongoing feedback loop ensured alignment with user needs and project goals.

Key technical achievements include the introduction of on-demand segment rendering, which prevents performance issues by avoiding the need to render all segments at once; the implementation of customizable tokenization to handle special character cases; an improved graphical user interface; enhanced support for metadata editing and fine-grained segment operations (such as resizing via drag actions); and a collaborative writing mode that synchronizes annotation states across multiple application instances.

Current efforts are largely focused on bug fixes and iterative improvements based on user feedback. Future plans involve introducing more advanced functionality and conducting

larger-scale testing to draw statistically meaningful conclusions about user interactions and feature effectiveness.

This thesis contributes not only to the technical and UX-oriented advancement of the platform but also to its broader mission within the JARIN project—namely, supporting the efficient and accurate transcription of Czech language data for academic research. Thanks to the platform developed within this thesis, annotation time in non-cooperative mode alone has been improved by **20%**, load time has been improved by **55%**, DOM content load speed increased dramatically, approaching constant time. The collaborative editing mode reduced annotation time by **55-65%**, making the research more effective. During the course of the thesis work, not only have I acquired a deep understanding of both the conceptual and practical underlying concepts behind the discussed topic but also have applied the learned principles in practical environments. This served as a reinforcement of the learned material and provided a significant understanding of the subject matter. I can certainly see this experience as a foundation for the possible future endeavors within the UX/UI field.

Bibliography

[1] *ASGI (Asynchronous Server Gateway Interface) Documentation* [online]. ASGI Team [cit. 2025-04-25]. Available at: https://asgi.readthedocs.io/.

[2] *The Complete Guide to Special Characters* [online]. Design Shack [cit. 2025-04-25]. Available at: https://designshack.net/articles/typography/the-complete-guide-to-special-characters/.

[3] *Django Documentation* [online]. Django Software Foundation [cit. 2025-04-25]. Available at: https://docs.djangoproject.com/en/5.2/.

[4] *Document Object Model (DOM) -– Web APIs* [online]. MDN Web Docs [cit. 2025-04-25]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model.

[5] *Etherpad: Collaborative Real-Time Editor* [online]. Etherpad [cit. 2025-04-25]. Available at: https://etherpad.org/.

[6] *Getting Started with WebSockets in TypeScript* [online]. Stackademic [cit. 2025-04-25]. Available at: https://blog.stackademic.com/getting-started-with-websockets-in-typescript-c48c5519f7d4.

[7] *How to use Django with Daphne (ASGI server for Django)* [online]. Django Software Foundation [cit. 2025-04-25]. Available at: https://docs.djangoproject.com/en/5.2/howto/deployment/asgi/daphne/.

[8] *An Introduction to Color Theory and Color Palettes* [online]. CareerFoundry [cit. 2025-04-25]. Available at: https://careerfoundry.com/en/blog/ui-design/introduction-to-color-theory-and-color-palettes/.

[9] *Redux Fundamentals, Part 2: Concepts and Data Flow* [online]. Redux.js Official Documentation [cit. 2025-25-04]. Available at: https://redux.js.org/tutorials/fundamentals/part-2-concepts-data-flow.

[10] *Reselect: Selector library for Redux (GitHub repository)*. GitHub. Accessed: 2025-04-25. Available at: https://github.com/reduxjs/reselect.

[11] *The WebSocket API (WebSockets) - Web APIs* [online]. MDN Web Docs [cit. 2025-04-25]. Available at: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[12] Abramov, D. *The Case for Flux [React Europe talk]*. YouTube, 2015. Accessed: 2025-04-25. Available at: https://www.youtube.com/watch?v=xsSnOQynTHs.

[13] AL., T. et. *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly, 2020. ISBN 978-1492051961.

[14] AU YEUNG, J. Web Storage Made Simple with use-local-storage-state. *LogRocket Blog*. May 2020. Available at: https://blog.logrocket.com/web-storage-made-simple-use-local-storage-state/.

[15] BANKS, A. and PORCELLO, E. *Learning React: Modern Patterns for Developing React Apps*. O'Reilly Media, 2020. ISBN 978-1492051725.

[16] COMEAU, J. W. *Persisting React State in localStorage: Introducing the "useStickyState" hook*. 24. Feb 2020. Available at: https://www.joshwcomeau.com/react/persisting-react-state-in-localstorage/.

[17] DUGOVIČ, J. *Transcription and annotation components for web editor in React*. Brno, CZ, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Szőke, I., 2024 [cit. 2025-04-25]. Available at: https://www.vut.cz/en/students/final-thesis/detail/155761.

[18] FAN, H., LI, K., LI, X., SONG, T., ZHANG, W. et al. CoVSCode: A Novel Real-Time Collaborative Programming Environment for Lightweight IDE. *Applied Sciences*. 2019, vol. 9, no. 21, p. 4642. DOI: 10.3390/app9214642.

[19] KRUG, S. *Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability*. New Riders, 2009. ISBN 978-0321657299.

[20] KRUG, S. *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*. New Riders, 2013. ISBN 978-0321965516.

[21] LINDLEY, C. *DOM Enlightenment*. O'Reilly Media, 2013. ISBN 978-1449342845.

[22] MARSH, J. *UX for Beginners: A Crash Course in 100 Short Lessons*. O'Reilly, 2016. ISBN 978-1-491-91268-3.

[23] MOJEED, I. Using localStorage with React Hooks. *LogRocket Blog*. Mar 2024. Available at: https://blog.logrocket.com/using-localstorage-react-hooks/.

[24] NECULA, S. Exploring The Model-View-Controller (MVC) Architecture: A Broad Analysis of Market and Technological Applications. *Preprints*. April 2024. Preprint, not peer-reviewed. Available at: https://www.preprints.org/manuscript/202404.1860/v1.

[25] PATEL, H. A Beginner's Guide to TypeScript: Understanding Types, Interfaces, & More – Part 1. *200OK Solutions Blog*. Dec 2024. Available at: https://200oksolutions.com/blog/a-beginners-guide-to-typescript-understanding-types-interfaces-more-part-1/.

[26] REACT DOJO. *Migrating From useState to useLocalStorage: A Journey to Persisting State* [Bits and Pieces (Medium)]. 13. Mar 2025. Available at: https://blog.bitsrc.io/migrating-from-usestate-to-uselocalstorage-a-journey-to-persisting-state-b07b3c185450.

[27] Singh, T. S. and Kaur, H. Review Paper on Django Web Development. *International Journal of Innovative Research in Technology*. 2023, vol. 10, no. 1, p. 1095–1099. ISSN 2349-6002. Accessed: 2025-05-01. Available at: https://ijirt.org/publishedpaper/IJIRT160728_PAPER.pdf.

[28] TypeScript Team. *The TypeScript Handbook*. 2024. Microsoft Documentation. Available at: https://www.typescriptlang.org/docs/handbook/intro.html.

[29] TypeScript Team. *TypeScript Handbook: Structural Type System*. 2024. Microsoft Documentation. Available at: https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html.

[30] Virginia, U. of. *Django and MVC – An Introduction to Software Engineering* [https://www.cs3240.org/modules/architecture/Django-and-MVC/]. N.d. Accessed: 2025-05-01.

[31] Wieruch, R. *Local Storage in React*. 2022. Available at: https://www.robinwieruch.de/local-storage-react/.

Appendix A

General User Feedback

The below feedback batches were collected at the Czech Academy of Sciences.

Batch A (January)

Below the feedback from the Academy of Sciences taken in January 2025:

> Highlight the control bar (`Play`, `Stop`, etc.) with color so that it's more intuitive. Currently, it's slightly overlapped by the scrollbar at the top.
>
> Default browser sliders are rendered in Chrome/Edge instead of custom ones.
>
> Add a notification confirming that saving was successful.
>
> Adding another way to zoom in the audio other than clicking the `+/-` buttons in the `Zoom` section. Users expect to be able to hold `Ctrl` and scroll with the mouse wheel, since that is the standard in other audio-related software. Clicking is very inefficient when they need to zoom in on a small section and quickly zoom out again. Keyboard shortcuts need to be added – e.g., `Ctrl` + scroll should zoom the audio (currently, it zooms the whole page).
>
> When zooming out during work on a segment, the waveform shifts, and the user has to scroll back to where they were. The waveform could stay centered on the area they're editing. The currently annotated audio segment should be highlighted. The playback indicator sometimes jumps to the start of the audio (unclear what triggers this) – this shouldn't happen.
>
> In Google Chrome, typing a number starting with 1–6 in the `Speed` input field causes a blank white screen. The user has to refresh the page, which returns them to the list of recordings. For example, 9% works, but 59% can't be set. Also, once the speed is changed, the user can't revert to 100%. Speed-up option needs to be introduced too.

Batch B (January)

> The user expects that clicking the waveform would also move focus to it so they can control playback with the spacebar. But if they previously had the cursor in a different input (e.g., speed or transcript), clicking the waveform doesn't transfer focus, and the spacebar just inserts a space. The user has to click outside the input field first – this should be improved.

Blank lines in the transcript field can't be deleted – once added, the user can't remove them. The box height remains expanded and can only be resized manually using the small triangle in the bottom right corner. Ideally, the box should resize automatically based on content.

Metadata deletion issue: When deleting content metadata covering a region that also includes other metadata blocks, both are deleted – but the labels remain visible on the segments. Afterwards, the user can no longer access or remove these metadata labels, as they disappear from the right-hand column.

Typing lag: Typing in the transcript field occasionally lags – there's a delay when typing or deleting characters. Metadata actions also suffer from lag – selecting start/end segments, searching metadata, confirming actions.

Batch C (February)

Add keyboard shortcuts for audio navigation while editing text (e.g., using `Ctrl` + arrow keys to jump in audio). Arrow keys don't work while the cursor is inside the text field.

When playback is stopped inside a segment, pressing Play resumes from where it stopped. It should instead restart from the beginning of the segment.

Users reported not being able to modify the end of a segment marked with certain metadata.

Allow tagging of transcriptions, e.g., `Reader's Edition`.

Content metadata doesn't display after saving; also, the title next to the transcript isn't showing. Content metadata loading is buggy in the editor.

In the recording filter, the `transcript` field should offer pre-filled options (instead of plain text).

Add Czech quotation marks (`„ "`) to the special characters menu.

Users have suggested semi-automatic segmentation: introducing a button for automatically creating the next segment would ensure smooth transitions between segments — i.e., the end of the previous segment aligns with the start of the next. Should still allow for manual editing on both sides.

Batch D (February)

Speaker labeling: When multiple speakers are present in the conversation, it's disruptive to automatically assign `Speaker A` when a new segment is created. Instead, the speaker of the previous segment should be assigned.

When the transcript spans multiple lines and one line is deleted, an empty space remains — it disappears only after returning to the list of jobs and reopening the recording.

Users expressed the need to add an option to increase/stretch the height of the audio waveform. Sometimes the waveform is so flat that at first glance it seems like there's no sound at all.

Disabling segment overlapping would be beneficial in cases when two speakers talk over each other.

Being able to play a selected segment in a loop would be useful.

Users would welcome the ability to select part of a recording without creating a segment. Sometimes it would be useful to figure out what's being said in that section. Then there should be an option to delete or heavily modify the segment. If a user is satisfied with the selection, they could press `Enter` to turn it into a segment.

No notification gets produced when changes are saved.