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

TOOL FOR SIMULTANEOUS EDITING OF MULTIPLE FILES WITH SUBTITLES

NÁSTROJ PRO SOUČASNOU EDITACI VÍCE SOUBORŮ S TITULKY

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **LUKÁŠ CHMELO**
AUTOR PRÁCE

SUPERVISOR **Ing. TOMÁŠ MILET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

156822

Institut:	Department of Computer Graphics and Multimedia (DCGM)
Student:	**Chmelo Lukáš**
Programme:	Information Technology
Title:	**Tool for Simultaneous Editing of Multiple Subtitle Files**
Category:	Computer Graphics
Academic year:	2023/24

Assignment:

1. Study subtitle formats and cross-platform user interface application development. Study video formats and ways to visualize an audio track. Explore existing subtitle editors.
2. Design an application that allows simultaneous editing of subtitles for multiple variants of the same video (containing cuts and inserted scenes). The goal is a tool that clearly displays aligned video tracks and subtitles and allows user to edit all subtitle files at the same time.
3. Implement the designed application so that it can be easily run on all common operating systems (Linux, Windows, ...). The application will allow user to check spelling and to edit the timing of the subtitles.
4. Continuously test the app and modify the design based on user feedback.
5. Review the final application, suggest future additions, publish the app and source codes and create a demonstration video.

Literature:

- Alan V. Oppenheim, Ronald W. Schafer. Discrete-Time Signal Processing. Prentice Hall. p. 1. ISBN 0-13-216771-9. 1989.
- Diaz-Cintas, Jorge. Subtitling: theory, practice and research. The Routledge Handbook of Translation Studies (pp.285-299). RoutledgeEditors: Carmen Millán, Francesca Bartrina. January 2012.

Requirements for the semestral defence:
A prototype able to edit multiple subtitle files with visualization of audio tracks and subtitle timing. Thirty pages of technical report.

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

Supervisor:	**Milet Tomáš, Ing., Ph.D.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	15.5.2024

Abstract

The aim of this work is to create a multi-platform tool designed for simultaneous editing of subtitle files in several video versions. The application enables simultaneous creation and editing of subtitles for different variations of one video, which contains different cuts and inserted scenes, thus solving a significant challenge in editing subtitles. Key features of the app include spell checking and speech timing editing features. Another important focus of this project is user-friendliness and cross-platform compatibility, which ensures ease of use across common operating systems.

Abstrakt

Cieľom tejto práce je vytvoriť multiplatformový nástroj určený na súčasnú úpravu súborov titulkov vo viacerých verziách videa. Aplikácia umožňuje súčasnú tvorbu a úpravu titulkov pre rôzne variácie jedného videa, ktoré obsahuje rôzne strihy a vložené scény, čím rieši významnú výzvu pri úprave titulkov. Medzi kľúčové funkcie aplikácie patrí kontrola pravopisu a funkcie úpravy časovania reči. Ďalším dôležitým zameraním tohto projektu je užívateľská prívetivosť a kompatibilita medzi platformami, čo zaisťuje jednoduché používanie naprieč bežnými operačnými systémami.

Keywords

cross-platform, audio, signal, video, subtitle, subtitle editing, subtitle formats, video formats, audio conversion, codec, waveform, correlation, DTW, MFCC, FFT, Javascript, ffmpeg, Electron

Klíčová slova

cross-platform, audio, signál, video, titulky, úprava titulkov, formáty titulkov, formáty videa, konverzia zvuku, kodek, vlnová forma, korelácia, DTW, MFCC, FFT, Javascript, ffmpeg, Electron

Reference

CHMELO, Lukáš. *Tool for simultaneous editing of multiple files with subtitles.* Brno, 2024. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Tomáš Milet, Ph.D.

Tool for simultaneous editing of multiple files with subtitles

Declaration

I declare that I have written this bachelor's thesis independently, under the supervision of Mr. Tomáš Milet. I have listed all the literary sources, publications, and other references used in this thesis.

. . . .
Lukáš Chmelo
May 16, 2024

Acknowledgements

I would like to express my gratitude to my supervisor, Mr. Tomáš Milet, who provided me with professional assistance and guidance throughout the project.

Contents

Chapter 1

Introduction

In today's digital age, audiovisual content plays a crucial role in various fields such as entertainment, education, and information dissemination. Subtitles serve as a key component in making videos accessible to a wide range of audiences. However, the manual process of aligning subtitles with the corresponding audio or video can be time-consuming and tedious. Addressing this problem involves developing a robust and efficient tool for automatic subtitle alignment. The concept of automatic subtitling is relatively new, enabled by the rise of artificial intelligence (AI). Instead of relying on AI, the implementation of this tool leverages algorithms such as MFCC and Cross Correlation to handle the required subtitle alignments effectively. This approach simplifies the process while still providing accurate results.

The main motivation of this work is to simplify the editing of subtitles for videos that are different versions of each other, meaning at least one of them contains extra scenes that the other doesn't. The videos contain many common parts, which implies that they will have common subtitles. It would be very useful if these subtitles could be edited concurrently, which is precisely what this tool does. The input video files remain unchanged, but changes in the input subtitle files are primarily outputs of this tool. This tool explicitly allows for creating new subtitles from scratch as well.

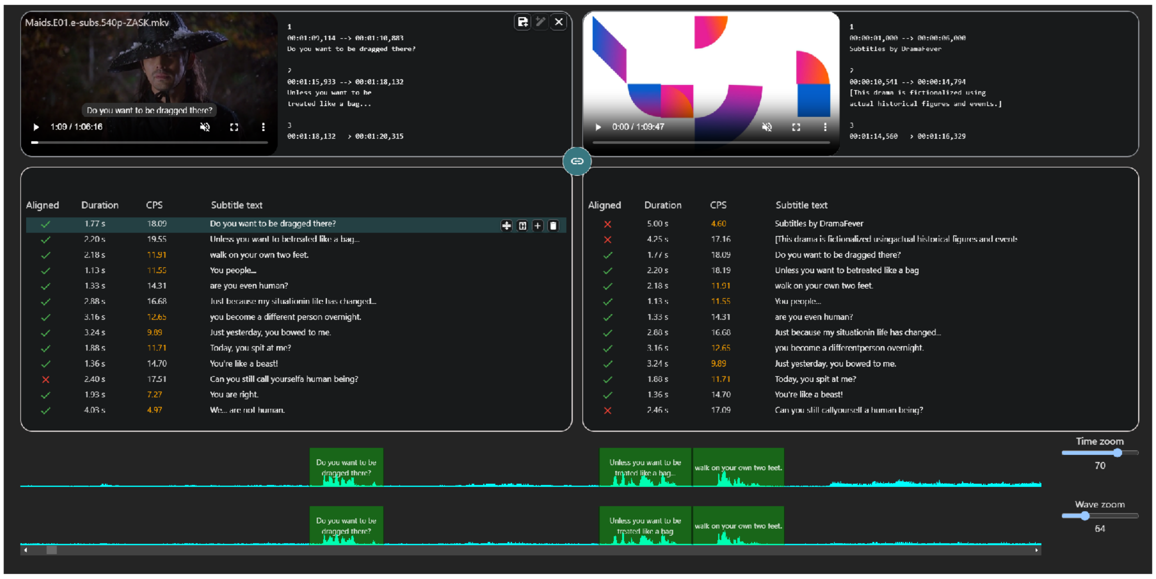
Simply put, the goal is to single-handedly edit multiple subtitles in one go. It may seem straightforward, but this is no easy task as it needs to be broken down into several steps. Firstly, the tool needs to determine what scenes one video contains that the other doesn't, which is done by comparing audio signals from the videos by converting video files to audio files (.wav, wavfile), extracting MFCC (Mel-frequency cepstral coefficients), and subsequent mutual comparison with cross-correlation of audio signals from the mentioned audio files. Secondly, it is important to visualize these alignments for the user, which is ensured in the user interface of this tool. Lastly, the user can concurrently edit aligned subtitles inside this tool with the help of keyboard shortcuts designed to enhance productivity.

The theoretical foundation of this work involves various technologies and methods. For instance, 2.1.4 and 2.1.4 explore cross-platform desktop application development frameworks, essential for creating a user-friendly interface. The discussion of subtitle formats such as SRT, WebVTT, and SubStation Alpha is integral, with specifics illustrated by 2.2. Audio-visual signal processing techniques, including waveform analysis (2.3.2) and the usage of Wavesurfer.js (2.3.2), are crucial for the accurate synchronization of subtitles. In addition, the application leverages powerful tools like FFmpeg for various functionalities. This includes transcoding (2.4.4), as depicted in 2.2, and subtitle extraction (2.4.4). Audio processing techniques, such as Dynamic Time Warping (DTW), Mel-frequency cepstral coefficients (MFCC), and Fast Fourier Transform (FFT), are essential for speech-to-text conversion (2.5.2) and are discussed in 2.6.2. The solution design chapter outlines the application's objectives and user interface design (3.1), supported by use cases (3.2). The technical stack and overall architecture are detailed, as seen in 3.5 and 3.6. The implementation chapter dives into the specifics of the application's components and modules (4.1.2), including the video player (4.3), subtitle table (4.4), and waveform handling (4.1.7). Back-end functionalities such as conversion and alignment (4.2.2, 4.2.3) are crucial for maintaining subtitle synchronization across different video versions.

1.1 Overview of the application

The result of this work is an application called **Subtitle Sync Editor**, designed for synchronizing subtitles with video content.

Subtitle Sync Editor is implemented using Electron, Vue3, JavaScript, and SCSS. This choice ensures compatibility across multiple platforms (Windows, Linux, Mac-OS) and provides a user-friendly graphical interface (Figure 1.1), allowing creators and translators to easily use the tool for subtitle alignment. The alignment process utilizes two main algorithms: cross-correlation and MFCCs, as detailed in 2.6.2.


Maids.E01.e-subs.540p-ZASK.mkv
Do you want to be dragged there?
1:09 / 1:06:16
0:00 / 1:09:47
Do you want to be dragged there?
Unless you want to be treated like a bag...
walk on your own two feet.
Unless you want to be treated like a bag
Time zoom
70
Wave zoom
64

```
1
00:01:09,114 --> 00:01:10,883
Do you want to be dragged there?

2
00:01:15,933 --> 00:01:18,132
Unless you want to be
treated like a bag...

3
00:01:18,132  > 00:01:20,315
```

```
1
00:00:01,000 --> 00:00:06,000
Subtitles by DramaFever

2
00:00:10,541 --> 00:00:14,794
[This drama is fictionalized using
actual historical figures and events.]

3
00:01:14,560  > 00:01:16,329
```

Aligned	Duration	CPS	Subtitle text
✓	1.77 s	18.09	Do you want to be dragged there?
✓	2.20 s	19.55	Unless you want to betreated like a bag...
✓	2.18 s	11.91	walk on your own two feet.
✓	1.13 s	11.55	You people...
✓	1.33 s	14.31	are you even human?
✓	2.88 s	16.68	Just because my situationin life has changed...
✓	3.16 s	12.65	you become a different person overnight.
✓	3.24 s	9.89	Just yesterday, you bowed to me.
✓	1.88 s	11.71	Today, you spit at me?
✓	1.36 s	14.70	You're like a beast!
✗	2.40 s	17.51	Can you still call yourselfa human being?
✓	1.93 s	7.27	You are right.
✓	4.03 s	4.97	We... are not human.

Aligned	Duration	CPS	Subtitle text
✗	5.00 s	4.60	Subtitles by DramaFever
✗	4.25 s	17.16	[This drama is fictionalized usingactual historical figures and events
✓	1.77 s	18.09	Do you want to be dragged there?
✓	2.20 s	18.19	Unless you want to betreated like a bag
✓	2.18 s	11.91	walk on your own two feet.
✓	1.13 s	11.55	You people...
✓	1.33 s	14.31	are you even human?
✓	2.88 s	16.68	Just because my situationin life has changed...
✓	3.16 s	12.65	you become a differentperson overnight.
✓	3.24 s	9.89	Just yesterday, you bowed to me.
✓	1.88 s	11.71	Today, you spit at me?
✓	1.36 s	14.70	You're like a beast!
✗	2.46 s	17.09	Can you still callyourself a human being?

Figure 1.1: Subtitle Sync Editor

The main window of the application shown in figure 1.1 is structured to enhance usability. The video players are positioned at the top, enabling users to view the video content and make precise adjustments to subtitle timing. Below the video players are the subtitle tables, which provide an organized layout for managing and editing subtitle text. At the bottom of the window, the audio waveform is displayed, facilitating accurate alignment of subtitles with the audio track.

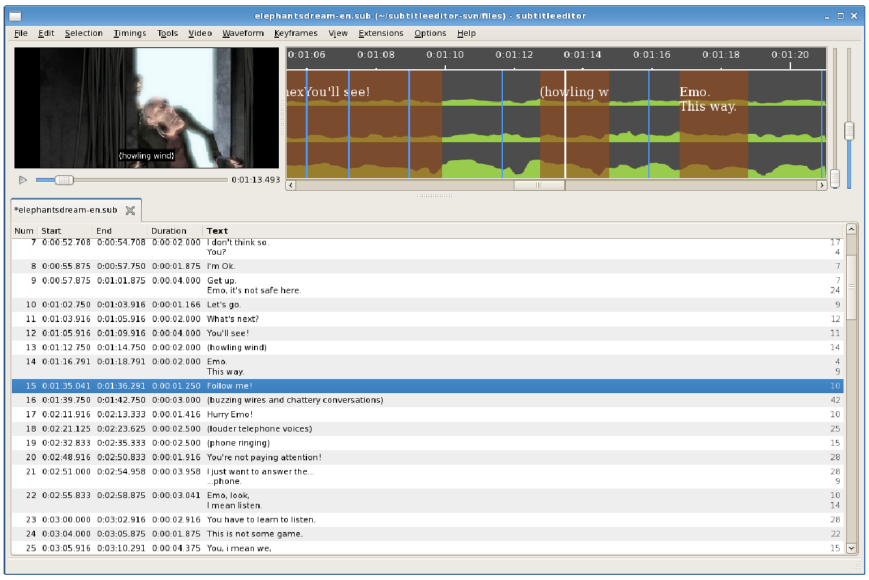
Further details about the user interface can be found in sections 3.3 and 4.1.4.

1.2 State of similar solutions

This section describes applications or tools that served as inspiration or competition for the development. Research has shown that an application that directly allows simultaneous editing of multiple subtitle files does not yet exist. Therefore, it is appropriate to discuss applications that address the basic issues of subtitle editing in a single video file, not the simultaneous solutions to this particular problem. The aim of the survey was also to identify the shortcomings of similar solutions, which the final application seeks to avoid.

1.2.1 Subtitle Editor

The Subtitle Editor tool, designed for Linux, was the main inspiration for the target functions and design of the final application. This tool allows the creation, editing, and positioning of subtitles, including setting their text, time, and length. All changes to the subtitles are visible in the waveform window (rectangular area for speech) and are automatically displayed during video playback. An illustration of the Subtitle Editor application interface can be found in figure 1.2.



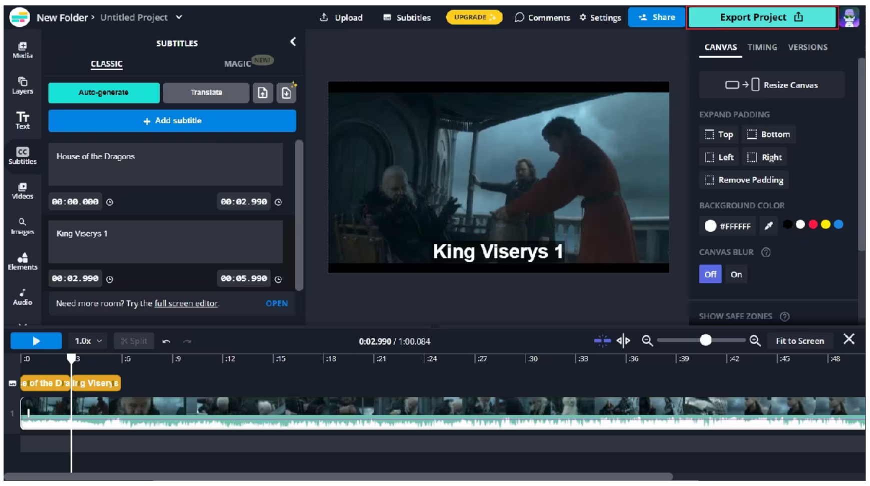
Num	Start	End	Duration	Text	
7	0:00:52.708	0:00:54.708	0:00:02.000	I don't think so. You?	17 4
8	0:00:55.875	0:00:57.750	0:00:01.875	I'm Ok.	7
9	0:00:57.875	0:01:01.875	0:00:04.000	Get up. Emo, it's not safe here.	7 24
10	0:01:02.750	0:01:03.916	0:00:01.166	Let's go.	9
11	0:01:03.916	0:01:05.916	0:00:02.000	What's next?	12
12	0:01:05.916	0:01:09.916	0:00:04.000	You'll see!	11
13	0:01:12.750	0:01:14.750	0:00:02.000	(howling wind)	14
14	0:01:16.791	0:01:18.791	0:00:02.000	Emo. This way.	4 9
15	0:01:35.041	0:01:36.291	0:00:01.250	Follow me!	10
16	0:01:39.750	0:01:42.750	0:00:03.000	(buzzing wires and chattery conversations)	42
17	0:02:11.916	0:02:13.333	0:00:01.416	Hurry Emo!	10
18	0:02:21.125	0:02:23.625	0:00:02.500	(louder telephone voices)	25
19	0:02:32.833	0:02:35.333	0:00:02.500	(phone ringing)	15
20	0:02:48.916	0:02:50.833	0:00:01.916	You're not paying attention!	28
21	0:02:51.000	0:02:54.958	0:00:03.958	I just want to answer the... ...phone.	28 9
22	0:02:55.833	0:02:58.875	0:00:03.041	Emo, look, I mean listen.	10 14
23	0:03:00.000	0:03:02.916	0:00:02.916	You have to learn to listen.	28
24	0:03:04.000	0:03:05.875	0:00:01.875	This is not some game.	22
25	0:03:05.916	0:03:10.291	0:00:04.375	You, i mean we,	15

Figure 1.2: Subtitle Editor

Moreover, the application is not supported on newer operating systems, such as versions of Ubuntu higher than 20.04 or other Linux distributions, which significantly limits its availability and usability on newer hardware or updated environments. This limitation can be a significant obstacle for users who prefer or need to use the latest technological solutions.

1.2.2 Kapwing

Kapwing is a web tool that offers a wide range of subtitle editing features, including the ability to customize fonts, styles, and subtitle positions. Additionally, it provides space for collaboration and video editing tools. It's user interface while editing subtitles is captured in figure 1.3.



House of the Dragons
00:00.000 00:02.990
King Viserys 1
00:02.990 00:05.990
Need more room? Try the full screen editor. OPEN
0:02.990 / 1:00.084

Figure 1.3: Editing subtitles in Kapwing

Despite its versatility, the tool has several limitations: it only supports the SRT format, the free version has various restrictions, users have reported issues with video processing and technical errors, and the video quality in the free version is low. A unique feature of Kapwing is its focus on simultaneous editing of multiple files, which sets it apart from other similar tools.

To use the simultaneous editing feature in Kapwing, you can simply add multiple video and subtitle files by clicking the „Upload“ button or dragging files directly into the tool's interface. Each file can be opened in a separate tab or window within the project. You can switch between files and make necessary edits, such as trimming videos, adding or editing subtitles. While working on individual files, Kapwing automatically synchronizes changes between them, meaning that edits in one file can immediately reflect in other files if the project's context requires it.

Kapwing's simultaneous editing feature allows users to work on different aspects of the project within individual files simultaneously, but it is important to understand that this tool does not support simultaneous creation of subtitles for multiple videos at once. Instead, it allows for simultaneous saving and synchronization of changes across multiple files, meaning that once a user makes an edit in one file, these changes can automatically apply or be reflected in other open files. This feature is very useful for efficient coordination and updating of projects where consistency needs to be maintained across multiple documents or media files, but unlike the final application, Kapwing does not allow direct simultaneous editing of subtitles for multiple videos at the same time.

1.2.3 Subtitle Workshop

Subtitle Workshop is a subtitle tool that stands out for its ease of installation on Windows, Linux, and OS X operating systems. It allows quick adjustment of subtitle duration, line editing, and spell checking, providing users with the basic tools needed for efficient subtitle work.

However, some users have reported that 64-bit versions of the program tend to crash frequently, and overall the software is considered slow and prone to errors. Another significant limitation is that Subtitle Workshop does not support the MKV format, which is often used for high-quality video files for which subtitles are created.

In terms of subtitle editing, Subtitle Workshop's features are similar to other tools on the market, but this tool also lacks the ability to simultaneously edit multiple files or direct simultaneous editing of subtitles for multiple videos at the same time.

1.2.4 Aegisub

Aegisub is a cross-platform subtitle tool that provides a wide range of features for synchronizing subtitles with audio. It allows users to see real playback time, includes text styling tools, a translation assistant, and displays a waveform, making it easier to accurately place subtitles according to the audio recording.

However, Aegisub can be complex for users without experience, as its interface and functionalities require a certain level of technical knowledge. Additionally, this tool does not support direct saving in the SRT format when certain features are used, which can be limiting for users who need this standard subtitle format.

Furthermore, Aegisub does not natively process multiple video files or subtitle files simultaneously within the same project.

Chapter 2

Theory

2.1 Cross-platform desktop application development

The development of desktop applications that operate across multiple operating systems presents unique challenges and opportunities for software engineers. This section aims to provide a foundational understanding of desktop applications, some of the key considerations in their cross-platform development, various frameworks and tools that enable the creation of robust, efficient, and versatile desktop applications that are compatible with Windows, macOS, Linux, and other platforms. The approaches and methodologies discussed here are inspired by insights from Paresh Kapuriya's article about desktop application development [6].

2.1.1 Desktop application and cross-platform application

Based on definition in the article by V2 Cloud [16], a desktop application is a dedicated software program designed to run on a standalone computer, enabling end-users to execute specific tasks. This application uses hardware resources of a computer to perform its functionality and is specifically designed to operate on a particular operating system.

On the other hand a cross-platform or multiplatform software is a type of application that works on various operating systems or devices, which are often called platforms as mentioned in article [17].

Common examples of desktop applications include Microsoft Office, Slack, VLC Media Player, Adobe Reader, WhatsApp, and WPS Office. To use these applications, downloading and installing them from their official website is required.

2.1.2 Desktop application development basics

The process known as desktop application development refers to the creation of software intended for use on desktop or laptop computers. Key considerations in this process include:

- Intended functionality of the application
- Target operating system
- Technology stack used

What is the intended functionality of the application

Desktop applications differ widely in terms of features, functionalities, and complexities. For instance, a gaming application requires a specific set of features designed for interactive entertainment. On the other hand, business-oriented applications might require custom features that fulfill organizational needs, such as the diverse functionalities provided by the Microsoft Office suite.

What is the target operating system

The development of desktop applications can vary significantly based on the operating system. Different operating systems like Linux, Windows, and macOS may require distinct technology stacks and coding structures. Additionally, compatibility with various libraries that may be essential for the application's functionality also needs to be ensured across these platforms. The development framework chosen is crucial as it should allow for easy adaptation to meet safety, security, and user experience needs across all intended platforms.

Use of programming languages

A variety of programming languages are available for the development of desktop applications. Common choices include HTML/CSS, Java, Python, C#, C++, Ruby, PHP, JavaScript, and TypeScript. The selection of programming languages typically depends on the specific requirements of the project and the developer's familiarity with the technologies. Moreover, modern frameworks like Xamarin and Qt offer extensive libraries and tools that support a broad range of languages and simplify the cross-platform development process.

2.1.3 Types of desktop software

Desktop software can be broadly categorized into several types, each serving distinct purposes and user needs. Understanding these categories is crucial for developers as it helps in choosing the right tools and approaches for development. Below are the primary types of desktop software commonly encountered.

Conventional application

Conventional desktop application software is developed to assist users in carrying out specific, diverse tasks effectively. Examples of application software include photo editing tools, accounting programs, workflow management applications, and multimedia players. These applications are designed to facilitate day-to-day tasks by providing functional solutions tailored to the user's needs.

Programming software

Programming software serves as a fundamental tool for developers, enabling them to create, test, debug, and maintain various software applications. This category includes integrated development environments (IDEs) like Visual Studio, IntelliJ IDEA, and Eclipse, which support various programming languages including Swift, C++, and PHP.

System software

System software is essential for managing computer hardware and provides a base for application software to function. It includes operating systems such as Linux, macOS, and Windows, which act as the intermediary layers between computer hardware and user applications, enabling the smooth execution of programs.

Browsers

Desktop browsers are applications used to navigate the internet. Common examples include Safari, Chrome, and Firefox, which allow users to access and interact with web content seamlessly. Although primarily associated with web development, desktop browsers also play a significant role in desktop application development, which is particularly evident in frameworks like Electron, as discussed in 2.1.4, where web technologies are utilized to build desktop applications.

2.1.4 Top development frameworks

The following frameworks provide a range of options for developers looking to build robust desktop applications tailored to specific operating environments and user needs. Each has its strengths and is chosen based on the requirements of the development project and the target audience.

Electron

Electron[1] is a leading framework for creating cross-platform desktop applications using web technologies like HTML, CSS, and JavaScript. It is renowned for its ability to integrate seamlessly across different operating systems and provides features such as automated updates, crash reporting, and a robust distribution system.

Technical advantages of using *electron.js*:

1. **Process Model** – Electron's process model, derived from `Chromium`[2], adopts a multi-process architecture where the main process functions similar to a backend in web development, managing core app functions and lifecycle. Renderer processes, on the other hand, acts like frontend, managing the user interface and user interactions, similar to web pages in a browser, each window in Electron runs its content in an isolated renderer process, managed by a central main process.

2. **Context Isolation** – Context isolation in Electron is a security feature that prevents scripts running in the renderer process from accessing Electron's internal APIs directly. This is similar to keeping users in their own secure, private space online, where they can't accidentally stumble into areas where they could cause damage. For example, without context isolation, a script on a web page could potentially access and manipulate the Electron APIs, leading to security vulnerabilities. With context isolation enabled, the webpage scripts and Electron's internal scripts operate in separate environments, communicating only through controlled channels like the `contextBridge`, which safely exposes specific functionalities to the web page.

[1]Introduction to electron - https://www.electronjs.org/docs/latest/

[2]Chromium - https://www.chromium.org/chromium-projects/

3. **Inter-Process Communication (IPC)** – In Electron, IPC is the mechanism that allows the render processes (frontend) and the main process (backend) to communicate, similar to how a client-side script (frontend) in a web application communicates with a server (backend). This is essential for actions that require access to deeper system functionalities from the render process, which are restricted for security reasons. For instance, if a render process needs to access files from the system, it sends a message to the main process using IPC, which then performs the action and returns the result.

Tauri

Tauri stands out as a versatile framework suitable for developing secure and efficient applications across major operating systems. It is a highly composable toolkit designed for creating desktop applications by using both Rust tools and web technologies like HTML, rendered in a `Webview`[3]. Unlike traditional frameworks that may include heavy runtimes, Tauri builds compile directly from Rust, as mentioned in Tauri's documentation[4]. This significantly reduces the app size because it uses the system's native `Webview` instead of creating a separate runtime. This approach enhances performance and reduces the binary size, making the reverse engineering of Tauri apps a non-trivial task due to the compilation to native code.

Core Architecture of *Tauri*:

- **Core Components** – The `Tauri crate` is at the heart of Tauri's architecture, integrating runtimes, utilities, and APIs. It manages the application's configuration via `tauri.conf.json`, and handles system interactions and updates.
- **Runtimes and Utilities** – `Tauri-runtime` serves as the intermediary layer between Tauri and various webview libraries, ensuring seamless integration and communication. Additionally, `Tauri-macros` and `Tauri-utils` provide essential tools for creating macros and parsing configuration files, ensuring efficient asset management and setup.
- **Bundling and Tooling** – Tauri includes a robust bundler that automatically detects and builds applications for the targeted platform. Its command-line tools, available in both Rust and JavaScript, streamline the development process and easily integrate into existing workflows.

Comparison of *Tauri* with *Electron*:

1. **Process Model** – Similar to Electron, described in 2.1.4, Tauri uses a multi-process architecture, but is optimized for efficiency by using the system's native `Webview` and not including a heavy runtime. This design significantly reduces resource consumption and application size.
2. **Security** – Tauri enhances security by minimizing the surface area for attacks through its architectural design and by leveraging Rust's inherent safety features. On the other

[3]Webview - https://tauri.app/v1/references/webview-versions/
[4]Tauri Architecture - https://tauri.app/v1/references/architecture/

hand, Electron relies on `Node.js` and `Chromium`, which may increase vulnerability due to their broader codebases.

3. **Inter-Process Communication (IPC)** – Tauri also supports IPC, described in 3, which enables secure and structured communication between the `Webview` (frontend) and the Rust-based backend. This setup ensures that system interactions are handled securely, respecting the application's integrity and security constraints.

Tauri's approach to application development, focusing on security, efficiency, and integration, offers a compelling alternative to Electron, especially for projects where binary size, performance, and security are of greatest importance.

Qt

Qt is a comprehensive framework designed for developing multi-platform desktop applications. It provides a broad range of tools and libraries that allow developers to deploy applications across various operating systems and embedded systems. Built primarily on C++, Qt offers both user interface (UI) and non-UI components.

As mentioned in Qt documentation [13], it supports the development of UI-driven applications through its extensive set of widgets and QML, a declarative language that enables sophisticated user interface layout with smooth animations and dynamic properties. It abstracts many platform-specific details, allowing code to be written once and deployed on multiple operating systems.

Challenges in Multi-platform Desktop Development with Qt:

1. **Complexity of Deployment** – Configuring and managing dependencies for different operating systems can be complex, with each platform having its unique requirements and limitations.

2. **UI Consistency** – Ensuring that the user interface appears and functions identically on all platforms can be challenging due to differences in native UI components and aesthetics.

3. **Learning Curve** – Qt uses C++, which may present a steeper learning curve compared to other high-level languages. The extensive nature of the Qt framework also demands a significant investment in learning its diverse functionalities.

4. **License Constraints** – Qt offers both open-source and commercial licenses. For commercial usage, licensing fees might be a consideration, particularly for startups and smaller companies.

Qt remains a popular choice for developers and aims to create robust cross-platform desktop applications, thanks to its comprehensive set of development tools and extensive library support. However, the challenges associated with multi-platform development, such as deployment complexity and ensuring UI consistency, require careful planning and expertise.

2.2 Subtitle Formats

This section, inspired by Emil Nikkhah's article on *Subtitle file formats: The most used and when to choose each one* [8], introduces various subtitle formats used in video applications, providing a technical overview and discussing their relevance in modern multimedia environments.

2.2.1 SubRip (SRT) Format

The most well-known and widely used subtitle format is the SubRip File Format (SRT). This format does not provide any styling or positioning adjustments inherently; however, various multimedia players such as VLC allow simple styling or formatting of .srt files using HTML tags like bold (), italics (), underline, and color. It is supported by most media players and several social platforms like Facebook and YouTube.

```
1
00:00:00,160 --> 00:00:06,200
Hello there!

2
00:00:06,600 --> 00:00:07,100
General Grievous.
```

Listing 2.2.1: Example of WebVTT subtitle format

In SRT format, subtitles have a straightforward structure, where each block contains:

- A serial number.
- Timecode, which marks start and end times in the format `hh:mm:ss,ms`.
- Text of the subtitle.
- Empty line between following dialogue blocks.

This format was chosen for its prevalence and simplicity, making it ideal for widespread use and accessibility across various platforms. Further discussion on SRT files is provided by authors Dick C.A. Bulterman, Jack Jansen, among others [3].

2.2.2 WebVTT Format (.vtt)

WebVTT (Web Video Text Tracks) is a modern subtitle format designed for the web, introduced with HTML5 to work alongside audio and video elements. This format offers comprehensive styling and positioning capabilities, which enhances the display of captions or subtitles to create more accessible and engaging multimedia experiences. Unlike the simpler SRT format, WebVTT allows for detailed formatting such as text color, font, and size adjustments directly in the captions as can be seen on listing 2.2.2.

```
WEBVTT

1
00:00:01.000 --> 00:00:04.000
<v Roger style="color:#e5e5e5; font-size:16px;">Welcome to our presentation!</v>

2
00:00:05.000 --> 00:00:10.000
<v Amy style="color:#00ff00; font-size:14px; font-style:italic;">Thank you for joining us today.</v>

3
00:00:11.000 --> 00:00:16.000
<v Sam style="color:#ff0000; font-size:20px; font-weight:bold;">Let's dive into the details.</v>
General Grievous.
```

Listing 2.2.2: Example of WebVTT subtitle format

WebVTT is widely supported across all major web browsers, including Google Chrome, Mozilla Firefox, and Safari. It is especially used by online video platforms like YouTube and Vimeo, becasue it is able to handle complex interactions and styles. Its robust features also make multimedia content better for viewers who are deaf or hard of hearing.

2.2.3 EBU-STL

EBU-STL is extensively used among broadcast channels and Video On Demand (VOD) services. This format allows encoding of subtitles, but it also imposes many rules on the customization of character numbers, colors, and positions. It is adaptable across various video formats, and is employed by the BBC and other major TV channels, making it ideal for broadcast companies and professionals due to its versatility beyond the limitations of the SRT format.

2.2.4 SubStation Alpha (SSA)

SubStation Alpha, or SSA, is an advanced subtitle format that offers extensive control over text parameters including font formatting, color, height, transparency, and placement. It is particularly popular within the anime community for its ability to create detailed, animated subtitles and lyrics for karaoke videos. Its more advanced version, ASS (Advanced SubStation Alpha), includes additional graphic and text features.

2.2.5 Timed Text Markup Language (TTML)

Timed Text Markup Language (TTML) is widely used by the television, broadcast, and VOD industry. Known for its high customizability, TTML excels in encoding and transposing text data into live video and audio streams, making it a prime choice for professional media environments that demand precise synchronization and detailed text formatting.

2.2.6 SUB

The SUB format is commonly utilized due to its straightforward approach, linking subtitles by frame number rather than time. This can lead to synchronization issues if the frame rate of the subtitle does not match the video. Unlike SRT, SUB offers no additional configuration options or support for advanced text effects, making it less reliable but very simple to use for basic subtitle needs.

2.3 Audio Visualisation and Processing

Audio visualization and processing involve the techniques and technologies used to visually represent and manipulate audio signals. This section explores key aspects of audio visualization, including waveform representation, and introduces algorithms, tools, and frameworks that facilitate audio processing. Most of the following information is based on the book *Continuous and Discrete Time Signals and Systems* [11].

2.3.1 Audio signal

Signals are crucial detectable quantities used to convey information about time-varying physical phenomena [11]. Mathematically, signals are modeled as functions of one or more independent variables, such as time, frequency, or spatial coordinates. This mathematical representation is fundamental in various fields of engineering, where signals play a pivotal role in system functionality.

In audio technology, signals represent sound waves captured over time, with time acting as the primary independent variable. Audio recording systems convert sound waves into electrical waveforms 2.3.2 which can be stored on various media such as magnetic tapes or digital discs, for example, by the waveform $y(t)$, which might be sinusoidal.

Signals can be classified based on how they are defined over time. A continuous-time (CT) signal is defined for all values of the independent variable, typically time (t). On the other hand, discrete-time (DT) signals are defined only at discrete points in time. These signals are sampled at specific intervals, which means they only provide signal values at those distinct time points. Audio signals are typically CT signals because they represent sound waves that vary continuously over time and have known magnitudes at every instant. DT signals are still essential in audio signal processing where continuous signals must be converted into a form that computers can handle.

2.3.2 Waveform Visualization

A waveform is a visual representation of an audio or electrical signal that charts the amplitude, or strength of a signal, over time, as defined in [15]. This type of visualization is crucial for understanding the dynamics and structure of an audio track.

The WAV file format 2.4.3, known for its uncompressed and high-quality audio output, is often used for waveform analysis because it preserves the original sound without any loss, providing a true representation of the audio's amplitude variations [4].

WaveSurfer.js

WaveSurfer.js[5] is a versatile tool for creating interactive waveforms directly in the web browser. It is particularly useful in applications developed with web technologies, such as projects using Electron as mentioned in 2.1.4, where it can be integrated to provide real-time audio waveform visualization.

WaveSurfer.js works by creating a canvas element and drawing the waveform on it. The waveform is generated by the Web Audio API[6], which is a high-level JavaScript API for processing and synthesizing audio in web applications. The API is built around the concept of an audio context, which represents a set of audio modules that are connected

[5] WaveSurfer - `https://wavesurfer.xyz/`

[6] Web Audio API - `https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`

together to process and generate sound. The audio context is created by instantiating the `AudioContext` object, which is the main entry point to the API. The audio context is used to create audio nodes, which are objects that represent audio sources, audio destinations, and audio processing modules. The audio nodes are connected together to form an audio graph, which represents the flow of audio data through the system. The audio graph is then processed by the audio context to generate sound.

Web Audio API uses some audio processing algorithms to analyze audio data such as FFT to analyze audio data and DFT to convert it to frequency domain, which will be described in the next section.

One of the key features of WaveSurfer.js is its high customizability. It allows developers to overlay HTML elements over the waveform at specific playback times, enhancing the interactivity and visual appeal of the waveform display. This makes it possible to implement custom controls, markers, and other interactive elements that improve user engagement and provide additional context to the audio being played.

2.3.3 Audio Signal Processing

Audio signal processing is a critical domain in digital signal processing that primarily deals with the manipulation and analysis of audio signals through digital methods. As discussed earlier, signals can be continuous or discrete in nature, and the transformations used in audio processing take these characteristics into account to perform various operations.

Discrete-Time Fourier Transform (DTFT)

The DTFT offers a means to represent discrete-time signals in the frequency domain. For a discrete-time sequence $x[k]$, the DTFT, denoted as $X(\omega)$, maps it into a continuous function in the frequency domain:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{2.1}$$

Breakdown of variables of equation 2.1:

- $X(\omega)$ is the DTFT of the sequence $x[n]$.
- ω is the frequency variable (radians per sample).
- n is the time index, ranging over all integers.
- j is the imaginary unit ($\sqrt{-1}$).

The transformation is particularly useful in analyzing the spectral components of a signal but poses computational challenges due to its continuous nature, making it less suitable for direct implementation on digital systems.

Discrete Fourier Transform (DFT)

The DFT addresses some of the computational challenges of the DTFT by discretizing the frequency domain. It transforms a sequence of N complex numbers into another sequence of N complex numbers, ideal for digital computations. The frequencies in DFT are given by $\Omega = 2\pi r/M$ for $r = 0, 1, \ldots, M-1$, where M is typically the length of the time-limited sequence $x[k]$ in following equation 2.2.

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j\frac{2\pi}{N}kn} \tag{2.2}$$

The problem with this algorithm is its speed, which is not ideal, especially when the length of the input signal is longer. Time complexity of the algorithm is quadratic $O(n^2)$, which is considered a relatively slow algorithm.

Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) addresses performance concerns inherent in the Discrete Fourier Transform (DFT). Essentially, the FFT is a more efficient version of the DFT that significantly increases the speed of the algorithm, even for shorter signals.

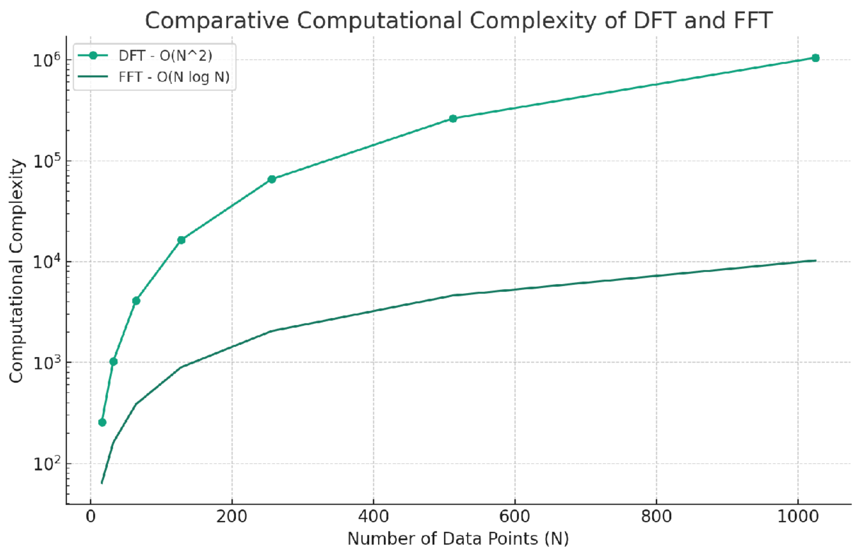


Figure 2.1: DFT vs FFT

The FFT is an algorithm that efficiently computes the DFT of a sequence, reducing the computational complexity to logarithmic $O(M \log M)$ as shown in 2.1. This significant reduction makes the FFT invaluable in digital signal processing for tasks such as convolution, correlation, and spectral analysis. The FFT is particularly advantageous in audio signal processing where large data sets are common, and timely processing is crucial.

The formula for the FFT is the same as for DFT, however, the process of the FFT involves breaking down the DFT into smaller DFTs of subsequences, typically exploiting symmetries and periodicities in the calculation process. The Cooley-Tukey algorithm is the most common FFT algorithm, which recursively divides the DFT into smaller DFTs. The elements of the input signal are divided into smaller parts, specifically by the parity of the index. This means the input array of elements is split into two arrays of size $N/2$ (where N is the size of the input signal). These arrays are then further divided into smaller arrays based on the same criteria, continuing until arrays of size 1 are achieved. Ultimately, the DFT of each smaller array is recursively computed up to the final result of the Fourier transformation of the input signal. This method utilizes the periodicity and symmetry of the complex exponential function, further discussed in *Discrete-time Signal Processing* [10].

2.4 Media Conversion

The process of media conversion is fundamental to managing and distributing digital content efficiently. It involves transforming multimedia files from one format to another, ensuring compatibility with various devices and platforms while maintaining quality. This section explores the essential components of media conversion, including codecs, video and audio formats, and the powerful functionalities offered by `FFMPEG`. Understanding these elements is crucial for optimizing the performance and usability of multimedia applications, particularly in scenarios involving large volumes of data and diverse user requirements.

2.4.1 Codecs and Encoding

Codecs are media compression technologies used for shrinking media files to sizes suitable for streaming and storage. This process is essential for content distributors handling large files. The term *codec* is an abbreviation for *coder-decoder* or *compressor-decompressor*, which applies algorithms to condense media into smaller versions, ensuring both delivery and storage.

The most commonly used media codec for video is `H.264` or Advanced Video Coding (AVC) and for audio Advanced Audio Coding (AAC). These codecs, such as `H.264/AVC` and `H.265/HEVC`, often have dual names because they were standardized by both the Moving Picture Experts Group (MPEG) and the International Telecommunication Union (ITU). Streaming codec expert Jan Ozer clarifies in his webinar [12] that this naming convention is typical for codecs recognized by multiple standards organizations, and it similarly applies to the Versatile Video Coding (VVC), also known as `H.266`.

2.4.2 Video Formats

A video file format is a type of file format for storing digital video data on a computer system. Video is almost always stored in compressed form to reduce the file size. A video file normally consists of a container (e.g. in the Matroska format) containing video data in a video coding format (e.g. VP9) alongside audio data in an audio coding format (e.g. Opus). The container can also contain synchronization information, subtitles, and metadata such as title etc. A video format specifies how to store the video and audio streams, metadata, and other, often optional, elements [5].

Several video formats are widely used today, each with its own advantages depending on the use case. In artice [2] Adobe highlights some of the most popular video formats that are commonly used in professional and consumer environments:

- **MP4 (MPEG-4 Part 14)** – Widely used for streaming over the internet due to its high compression and quality ratio.
- **MOV** – Developed by Apple, preferred for high-quality video editing due to its flexibility in encoding options.
- **AVI (Audio Video Interleave)** – Introduced by Microsoft, known for less compression, resulting in larger file sizes but higher quality.

- **WMV (Windows Media Video)** – Often used for streaming applications within the Windows ecosystem.
- **FLV (Flash Video)** – Primarily used for embedded video on the web, although less common since the decline of Adobe Flash.
- **AVCHD (Advanced Video Coding High Definition)** – Utilized primarily for digital recording and playback of high-definition video.

Each format serves distinct purposes and is chosen based on criteria such as compatibility, quality requirements, and distribution needs.

2.4.3 Audio Formats

An audio file format is a file format for storing digital audio data on a computer system. The format may be containerized, where each file holds multiple different types of data streams, or non-containerized, consisting of a single data stream in a format like WAV or MP3. Audio formats can be uncompressed, lossless, or compressed with lossy compression [4].

Several audio formats are widely recognized for their effectiveness in various applications, particularly in professional and consumer audio environments:

- **WAV** – Known for its uncompressed high-quality audio output, ideal for professional recording and editing.
- **MP3** – Highly popular for its balance of quality and file size, making it a standard choice for digital music distribution.
- **AAC** – Superior in compression and quality to MP3, commonly used for streaming and in devices like smartphones.
- **FLAC** – A lossless audio format that provides high-quality sound without any loss, perfect for audiophiles who require pristine audio.
- **ALAC** – Apple's version of FLAC, used primarily within its ecosystem to deliver lossless audio.

These formats are selected based on their specific attributes such as sound quality, file size, and compatibility with playback devices [1].

2.4.4 FFMPEG

FFMPEG is a versatile tool used widely in the multimedia field for its powerful media processing capabilities. Its functionality spans various aspects of media handling, including reading from numerous input sources, filtering, and transcoding to multiple output formats. Below is a structured explanation adapted from the official FFMPEG documentation[7].

FFMPEG functions as a universal media converter, capable of handling a wide array of media types, from regular files to live input sources such as network streams or recording devices. This robust tool reads input through the `-i` option and outputs via a straightforward URL format. Any command-line input not recognized as an option is treated as an output URL.

[7]FFMPEG - `https://ffmpeg.org/ffmpeg.html`

The software can process multiple input and output files simultaneously, supporting various stream types within each file, such as video, audio, subtitles, and data. The format of the container may limit the number of streams or types it can handle. Stream selection for the outputs can be automatic or manually controlled through the `-map` option. File and stream indices are used to reference inputs in command-line options, which are zero-based (e.g., the first file is indexed as 0).

Ordering of options is crucial, as they apply sequentially to the next specified file. Global options, like verbosity settings, are exceptions and should be specified at the beginning.

Transcoding Process

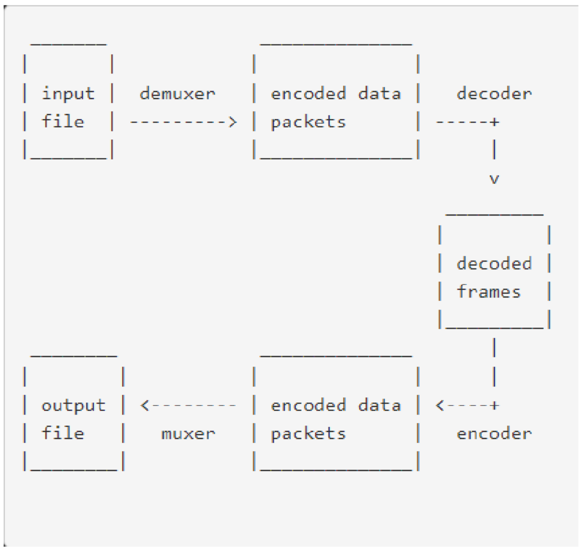


Figure 2.2: FFMPEG Transcoding process

The transcoding workflow as shown in Figure 2.2 involves several key steps, each using different components of the FFMPEG architecture:

- **Demuxing** – Initially, FFMPEG invokes the libavformat library to demux input files and extract packets of encoded data. This step is critical when handling multiple input files to maintain synchronization, which is achieved by tracking the lowest timestamp across active streams.
- **Decoding** – Encoded packets are then sent to the decoder. Depending on the user's selection, the stream may either be decoded or passed through via stream copying. The decoder's output is uncompressed frames (video, PCM audio, etc.).
- **Filtering** – Post-decoding, frames can be further processed through various filters, which modify or enhance the content before encoding.
- **Encoding** – The processed frames are re-encoded into the desired output format, creating new packets of encoded data.
- **Muxing** – Finally, these packets are sent to the muxer, which writes them to the output file, completing the transcoding process.

Example usages of FFMPEG

FFMPEG offers a multitude of command-line options for various media processing tasks. Here are some fundamental examples, including video format conversion, audio extraction, audio embedding, and frame rate modification.

```
ffmpeg -i input.avi -codec:v libx264 -codec:a aac output.mp4
```

Listing 2.4.1: Convert a video from one format to another(AVI to MP4)

Command shown in listing 2.4.1 reads the AVI file (`input.avi`), re-encodes the video using the H.264 video codec (`libx264`) and the audio using the AAC audio codec, and then writes the output to an MP4 file (`output.mp4`).

```
ffmpeg -i input.mp4 -vn -codec:a libmp3lame output.mp3
```

Listing 2.4.2: Extract audio stream from a video file and save as MP3

Command shown in listing 2.4.2 extracts the audio stream from a video file (`input.mp4`) and saves it as an MP3 file (`output.mp3`). The `-vn` option is used to skip the video processing, and `libmp3lame` specifies the MP3 encoder.

```
ffmpeg -i video.mp4 -i audio.mp3 -codec:v copy -codec:a aac -strict experimental output.mp4
```

Listing 2.4.3: Embed an audio file into a video file

Command shown in listing 2.4.3 embeds (or muxes) an audio file into a video file, replacing any existing audio. This command takes a video file (`video.mp4`) and an audio file (`audio.mp3`) and combines them into one output file (`output.mp4`), encoding the audio to AAC. The video stream is copied directly without re-encoding (`-codec:v copy`).

```
ffmpeg -i input.mp4 -filter:v fps=fps=30 output.mp4
```

Listing 2.4.4: Change the frame rate of a video file

Command shown in listing 2.4.4 changes the frame rate of a video file. This example sets the frame rate of the output video (`output.mp4`) to 30 frames per second using the `fps` filter.

Embedding subtitles to video

FFMPEG also provides a command that combines video and subtitle files into a single output with the subtitles encoded in a format compatible with most media players. After creating or editing subtitles, this functionality allows for the seamless integration of the subtitles with the video, ensuring that they can be displayed correctly across a wide range of devices and media players.

```
ffmpeg -i inputFilePath -i subtitleFilePath -c copy -c:s mov_text -map 0 -map 1 outputFilePath
```

Listing 2.4.5: Embedding subtitles to video using ffmpeg

Here is a breakdown of parameters used in listing command 2.4.5:

- `-i inputFilePath` specifies the path to the input video file.
- `-i subtitleFilePath` specifies the path to the subtitle file.
- `-c copy` instructs FFMPEG to copy all the streams (video, audio, etc.) from the input files without re-encoding them.
- `-c:s mov_text` sets the codec for the subtitle stream. Here, `mov_text` is used, which is a common format for MP4 files.
- `-map 0` and `-map 1` are used to map all streams from the first input file (video and possibly audio) and the subtitle stream from the second input file to the output file.
- `outputFilePath` is the path where the output file will be saved.

Extracting subtitles from video

Extracting subtitles from a video is a straightforward task efficiently managed by FFMPEG. This process is essential, particularly when compared to embedding subtitles, as it allows for further editing or modification of the subtitles separately from the video content.

```
ffmpeg -i inputFilePath -map 0:s:0 -c:s srt outputFilePath
```

Listing 2.4.6: Extracting subtitles from video using ffmpeg

Here is a breakdown of parameters used in listing command 2.4.6:

- `-i inputFilePath` specifies the video file from which to extract the subtitles.
- `-map 0:s:0` selects the first subtitle stream from the first file. Here, `0:s:0` indicates the first (0th) input file, subtitles streams (`s`), and the first subtitle stream in the list (`0`).
- `-c:s srt` specifies the codec for the subtitle, which in this case is `srt` or SubRip subtitle format.
- `outputFilePath` is the path where the extracted subtitle file will be saved, typically with an `.srt` extension.

2.5 Speech detection in audio

This section, inspired by article [9], explores techniques for speech detection by detecting loud sections and converting speech to text in audio, so that creation of subtitles becomes simpler. By identifying loud sections it can be estimated where some potential conversations might occur, which helps to pinpoint where subtitles might be needed. Meanwhile, speech-to-text technology can nearly automate subtitle generation by transcribing spoken words into text.

2.5.1 Loud Section Detection

Detecting loud sections in audio involves identifying parts of an audio signal where the volume exceeds a predefined threshold. This technique is crucial for applications like broadcasting and audio mixing, where maintaining audio levels within certain limits is necessary for quality control and regulatory compliance. Machine learning models, such as those reviewed in the Detection and Classification of Acoustic Scenes and Events (DCASE) competitions, often utilize features like Mel-Frequency Cepstral Coefficients (MFCC) and spectrograms to classify and analyze different sound events, including loudness anomalies.

2.5.2 Speech to Text

Speech-to-text technology converts spoken language into written text using advanced algorithms, primarily deep learning models like Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). This technology is integral to services such as real-time captioning, voice-driven search, and virtual assistants. The effectiveness of these systems often depends on the quality and diversity of the training data, which helps the model understand various phonetics, accents, and dialects.

2.6 Subtitle alignment

When working with multiple video files that are variations of each other (they include extra scenes or cutouts), aligning their audio waveforms for accurate subtitle creation is essential. This section suggests methods on how this can be done.

2.6.1 Text-based alignment

Using previously mentioned speech-to-text technology in 2.5.2, it can simplify the process of aligning subtitles across two or more video files. This method involves converting the speech from each video into text and then using algorithms to compare and synchronize the text based on similarities and timing.

Even without speech-to-text technology, this approach can still be effective if an editor has already begun creating subtitles manually. In such cases, the existing subtitles can be used as a base for further alignment and synchronization.

2.6.2 Signal-based alignment

Apart from using speech-to-text technology, there are other methods for aligning subtitles without relying on transcription. Signal-based alignment involves analyzing the audio signals of the videos to synchronize subtitles. This method can be particularly useful when speech-to-text technology is not viable due to language barriers, audio quality, or the complexity of implementing such technology. Signal-based approaches might include techniques like cross-correlation of audio waveforms or DTW, which will be discussed in further.

Cross Correlation

Cross correlation is a method used to measure the similarity between two signals by calculating the similarity of two series as a function of the displacement of one relative to the other. It involves comparing one signal to another to detect the presence of similar features or patterns. This technique is often referred to as sliding dot product or sliding inner-product.

Given two finite-length sequences $x[n]$ and $y[n]$, where n ranges from 0 to $N-1$ (assuming both sequences are of equal length for simplicity), the cross-correlation $r_{xy}[m]$ at lag m is defined as follows:

$$r_{xy}[m] = \begin{cases} \sum_{n=0}^{N-1-m} x[n+m] \cdot y[n] & \text{for } m \geq 0 \\ \sum_{n=0}^{N-1+m} x[n] \cdot y[n-m] & \text{for } m < 0 \end{cases} \tag{2.3}$$

Breakdown of variables of equation 2.3:

- $r_{xy}[m]$ represents the cross-correlation of sequences x and y at shift m.
- $x[n+m]$ is the $(n+m)$-th element of sequence x, and $y[n]$ is the n-th element of sequence y for $m \geq 0$.
- For $m < 0$, the roles of x and y are effectively reversed with an adjustment to the indices to maintain alignment.
- The sums are adjusted depending on the lag to ensure that the indices do not exceed the bounds of the sequences.

Cross-correlation, as implemented by the MATLAB function `xcorr(x, y)`, computes the similarity between two discrete-time sequences, x and y. This function calculates the degree of overlap between x and various shifted (lagged) versions of y, producing a series of values that measure the similarity at each shift. This is crucial for identifying the time delay between the signals, helping to determine how one sequence is displaced or shifts behind the other.

DTW

Dynamic Time Warping (DTW) is a non-linear algorithm crucial for aligning sequences that vary in time or speed. Originally developed for speech recognition, DTW has expanded into fields like robotics and meteorology due to its robustness in time series classification (TSC). Its adaptability across different domains underscores its significance and versatility as one of the most competitive algorithms in TSC.

Given two time series, Q and C, with lengths n and m respectively shown here 2.4:

$$Q = \{q_1, q_2, \ldots, q_n\}\, C = \{c_1, c_2, \ldots, c_m\} \tag{2.4}$$

DTW aligns these series by constructing a matrix $D(n, m)$, where each element $D(i, j)$ denotes the Euclidean distance squared between q_i and c_j:

$$ED\left(q_i, c_j\right) = \left(q_i - c_j\right)^2 \tag{2.5}$$

The algorithm seeks a warping path W that minimizes the total distance across the matrix while adhering to boundary, continuity, and monotonicity constraints. This path begins at $D(1, 1)$ and ends at $D(n, m)$, ensuring that the sequences are aligned from start to finish without reducing the path's length. The optimal path is the one that minimizes the sum of these squared distances, calculated using dynamic programming:

$$D(i, j) = d\left(q_i, c_j\right) + \min(D(i-1, j-1), D(i-1, j), D(i, j-1)) \tag{2.6}$$

MFCC

Mel-frequency cepstral coefficients (MFCC) capture the frequency spectrum of sound using various algorithms and adjustments to best simulate the functioning of the human ear and the subsequent processing of the signal by the brain. This method allows for efficiently capturing important aspects of sound in a smaller number of values (coefficients), ideally carrying the same amount of information as the original signal, which aids in various speech recognition applications. One step in extracting MFCC coefficients involves converting the frequency scale to the so-called Mel scale, which better reflects how humans can distinguish differences in various frequencies (the human ear has higher resolution at lower frequencies than at higher ones). The formula for converting hertz to mels is:

$$f_{\text{Mel}} = 2595 \log_{10}\left(1 + \frac{f}{700}\right) \tag{2.7}$$

In equation 2.7 the Mel frequency scale takes on a logarithmic form, with the formula taken from the work of authors K. Sreenivasa Rao and Manjunath K.E. [14]. Originally created for automatic speaker recognition, today they are also used in music information retrieval, as noted by the author M. Müller in „Information Retrieval for Music and Motion“ [7].

Chapter 3

Solution Design

This chapter includes set objectives and all tasks necessary to achieve these objectives. It explains how to apply the algorithms, technologies and other resources outlined in Chapter 2 to address the problem introduced in the thesis' introduction.

3.1 Objectives

In order to establish objectives, it is essential to consider the user's needs. Primarily, the user wants to comfortably input data (videos and subtitles), be able to edit the inputted data, have displayed immediate outputs with changes in multiple files at once after editing and be able to save these outputs, while also ensuring that all background and foreground processes are completed as quickly as possible.

- **Cross-platform Compatibility** – Utilize the cross-platform development tools outlined in Section 2.1 to ensure the application functions seamlessly across different operating systems.
- **Rich Features** – Design the application to enable users to efficiently edit multiple subtitle files concurrently across various video versions, enhancing productivity and streamlining the editing process.
- **Open Source** – Develop an engaging project that encourages contributions from the community. This involves writing readable code and integrating popular frameworks to foster collaboration.
- **User Experience** – Design a user interface that is attractive and responsive, drawing on the best features of existing popular applications. The interface should be intuitive, enhanced with tool-tips, icons, and animations to facilitate user interaction.
- **Performance** – The application should ensure high efficiency and responsiveness even when handling complex tasks, allowing users to execute multiple operations simultaneously without noticeable delays.
- **Tech Support** – Develop comprehensive educational materials, including technical documentation and video tutorials, to support developers and end-users in using and understanding the software effectively.

From the objectives above, the following tasks can be extracted to achieve them:

1. **Define use cases** – The project requires defining use cases to prioritize the development of the mentioned rich features.
2. **Design user interface** – In order to achieve the best user experience a user interface must be designed and implemented.
3. **Technology stack selection** – Choosing the appropriate technology stack is critical for ensuring cross-platform compatibility.
4. **Design application architecture** – Designing the application architecture is essential to maintain high performance in processing complex tasks.
5. **Implementation** – Implement designed user interface and implement the application following the designed architecture.
6. **Create tutorials** – Finally, creating support and educational materials is a crucial task to help both developers and end-users understand and effectively use the software.

These tasks collectively lay the groundwork for a robust and user-friendly software application that aligns with the outlined objectives. The following sections focus on the design tasks(1-4) from these tasks.

3.2 Use cases

This section is dedicated to identifying key features that are critical for the user to effectively manage input data, ensuring a smooth workflow. These features will enable users to make the necessary changes easily, leading to the desired output data. The focus on these essential functionalities is aiming to enhance the user's overall experience and efficiency.

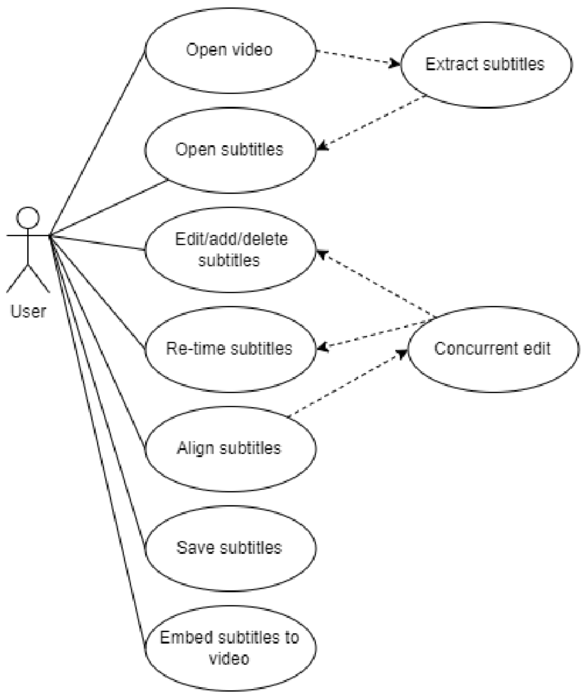


Figure 3.1: Use Case Diagram

Figure 3.1 shows use cases for the central actor, `user`, who engages in various actions related to subtitle management in video files:

- **Open video** – Opens a file dialog from which the user can select video file. Supported video formats include MP4 and MKV.
- **Open subtitles** – Also opens a file dialog from which the user can select subtitle file. Supported subtitle formats include SRT and VTT.
- **Extract subtitles** – Alternative to opening subtitles, when a video is opened and there are embedded subtitles within the video file. This allows for direct manipulation of existing subtitles without the need for separate subtitle files.
- **Edit/add/delete subtitles** – After gaining access to the subtitles, users have the flexibility to directly alter the text in the subtitle files, including the ability to add new subtitles or remove existing ones as needed. Editing will also include spell-check, which will work by highlighting any
- **Re-time subtitles** – The process of re-timing subtitles is essential for synchronizing the text with the corresponding video, ensuring that the subtitles appear at appropriate moments in alignment with the audio and visual elements.
- **Align subtitles** – Works by using advanced algorithms to locate segments of audio from one video file (beneath the subtitles) and match them with audio from another video file. This capability facilitates precise synchronization between two different video sources, each with their own subtitles.
- **Concurrent edit** – Following the alignment of subtitles and their corresponding audio tracks, this feature enables simultaneous editing of both subtitle files. This approach allows for cohesive updates and modifications across the two videos, enhancing efficiency and accuracy in subtitle editing
- **Save subtitles** – Allows users to save their edited subtitles in formats SRT and VTT for subsequent use or distribution.
- **Ebmed subtitles to video** – The opposite of extracting; this function embeds the edited subtitles back into the video, merging them permanently without the need for separate subtitle files.

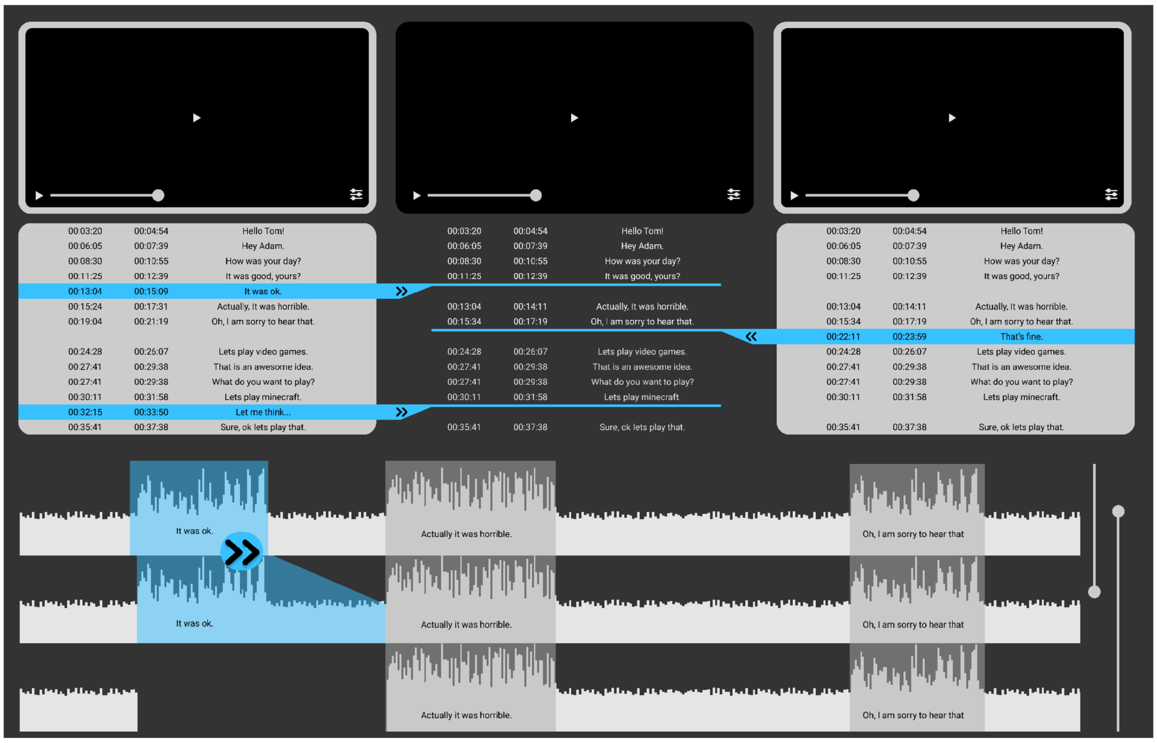
The features above are aiming to cater to the user's primary tasks, such as inputting data, editing content, and managing outputs efficiently.

3.3 User interface design

This section focuses on the design of the user interface, which is crucial for achieving efficient interaction between the user and the software. The user interface needs to be able to support all features mentioned in section 3.2. Also, as mentioned in 3.1, the ultimate goal is to create an environment where users can achieve their objectives effortlessly, enhancing their overall experience with the software.

3.3.1 Loaded state

Figure 3.2 illustrates the initial design of the application's user interface in a scenario where two video files and their corresponding subtitle files are open. The design prominently features video players at the top, with subtitle tables below each and audio waveforms further down, providing a comprehensive view of both the visual and textual content. The central video player, along with its accompanying subtitle table and waveform, serves a critical role in highlighting differences between the subtitles and audio signals. This central component was intended to make it evident which subtitles were absent or additional in either of the files.



00:03:20	00:04:54	Hello Tom!
00:06:05	00:07:39	Hey Adam.
00:08:30	00:10:55	How was your day?
00:11:25	00:12:39	It was good, yours?
00:13:04	00:15:09	It was ok.
00:15:24	00:17:31	Actually, It was horrible.
00:19:04	00:21:19	Oh, I am sorry to hear that.
00:24:28	00:26:07	Lets play video games.
00:27:41	00:29:38	That is an awesome idea.
00:27:41	00:29:38	What do you want to play?
00:30:11	00:31:58	Lets play minecraft.
00:32:15	00:33:50	Let me think...
00:35:41	00:37:38	Sure, ok lets play that.

00:03:20	00:04:54	Hello Tom!
00:06:05	00:07:39	Hey Adam.
00:08:30	00:10:55	How was your day?
00:11:25	00:12:39	It was good, yours?
00:13:04	00:14:11	Actually, It was horrible.
00:15:34	00:17:19	Oh, I am sorry to hear that.
00:24:28	00:26:07	Lets play video games.
00:27:41	00:29:38	That is an awesome idea.
00:27:41	00:29:38	What do you want to play?
00:30:11	00:31:58	Lets play minecraft.
00:35:41	00:37:38	Sure, ok lets play that.

00:03:20	00:04:54	Hello Tom!
00:06:05	00:07:39	Hey Adam.
00:08:30	00:10:55	How was your day?
00:11:25	00:12:39	It was good, yours?
00:13:04	00:14:11	Actually, It was horrible.
00:15:34	00:17:19	Oh, I am sorry to hear that.
00:22:11	00:23:59	That's fine.
00:24:28	00:26:07	Lets play video games.
00:27:41	00:29:38	That is an awesome idea.
00:27:41	00:29:38	What do you want to play?
00:30:11	00:31:58	Lets play minecraft.
00:35:41	00:37:38	Sure, ok lets play that.

It was ok.
Actually it was horrible.
Oh, I am sorry to hear that
It was ok.
Actually it was horrible.
Oh, I am sorry to hear that
Actually it was horrible.
Oh, I am sorry to hear that

Figure 3.2: UI design - loaded state

This layout also proposed an innovative feature for merging the two videos and their subtitles into a single file, which could be done by buttons with double arrows located on the waveform and at the edges of the subtitle tables. However, as detailed in section 4.1, the final user interface design underwent significant changes. The central comparison component was removed to simplify the interface and avoid potential confusion among users, as merging subtitles was not a primary function. This adjustment led to a more straightforward method of displaying comparisons, which aligned better with the user's needs and simplified the front-end implementation.

Despite these changes, the foundational structure of the user interface remained largely intact. The revised design still incorporates the core components initially planned—video players, subtitle tables, and waveforms—maintaining the essence of the original layout while enhancing usability and clarity for the end users.

3.3.2 Opening states

Figure 3.3 shows the same application layout but in scenarios where not all necessary files for aligning subtitles are opened.

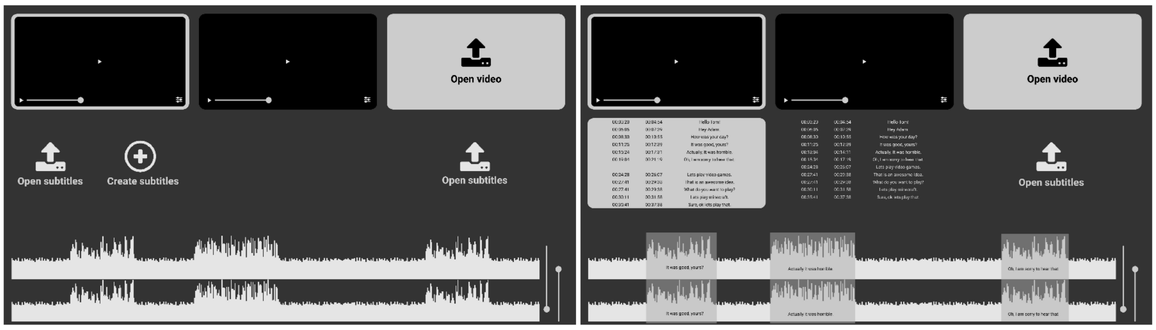


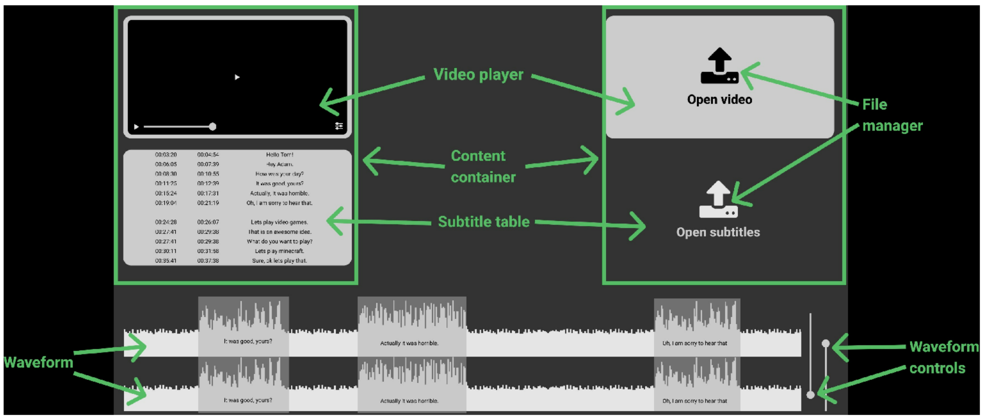
Figure 3.3: UI design - opening states

In the first scenario, shown on the left, only one video file is open. Users can open an existing subtitle file by clicking on the `Open subtitles` button, or they can start creating subtitles from scratch by selecting the `Create subtitles` button on the left panel of the application. Additionally, another video file can be opened by using the `Open video` button on the right panel.

The second scenario, depicted on the right, demonstrates what occurs when a subtitle file is loaded under its corresponding video. In this setup, the subtitles appear within the waveform display as editable regions. These regions can be moved around, allowing users to adjust the timing of the subtitles directly on the waveform. This visual representation helps in easily seeing and modifying the timing of subtitles to ensure they match the video's audio track.

3.3.3 Components

This subsection will break it down the user interface into components in order to fully address all functionalities of the user interface, which are supposed to support all features mentioned in section 3.2.



00:03:20	00:04:54	Hello Tom!
00:06:05	00:07:39	Hey Adam.
00:08:30	00:10:55	How was your day?
00:11:25	00:12:39	It was good, yours?
00:15:24	00:17:31	Actually, it was horrible.
00:19:04	00:21:19	Oh, I am sorry to hear that.
00:24:28	00:26:07	Lets play video games.
00:27:41	00:29:38	That is an awesome idea.
00:27:41	00:29:38	What do you want to play?
00:30:11	00:31:58	Lets play minecraft.
00:35:41	00:37:38	Sure, ok lets play that.

Figure 3.4: UI design - components

Figure 3.4 provides a detailed breakdown of the user interface, where each component is clearly identified using arrows and labels in green. Below is an overview of these components and their specific roles within the application:

- **Video player** – This component is essential for video playback. If no video is loaded, it displays a button that prompts the user to open a video via the file manager component. This ensures that users can easily load and play videos directly from the interface.
- **Subtitle table** – Displays all loaded subtitles and allows for comprehensive editing actions such as adding, deleting, splitting, merging, and duplicating subtitles. It also facilitates setting playback to match the selected subtitle. When no subtitle file is loaded, it too displays a button that directs users to open subtitles using the file manager component, mirroring the functionality of the video player. Keyboard shortcuts, detailed in 4.1.6, enhance usability by speeding up common tasks.
- **Content container** – Acts as a wrapper for the subtitle table and video player, integrating both components to synchronize video playback with the active subtitles, enhancing the viewing and editing experience.
- **File manager** – Features a button with a popup menu that lists recently opened files for quick access or allows the opening of new files. This component is crucial for managing file input and provides a streamlined way to access and load content.
- **Waveform** – Visualizes audio tracks from videos as waveforms and overlays subtitles as interactive regions within these waveforms. These regions update reactively based on changes in the subtitle table and can be manipulated directly for aligning or deleting subtitles. Additionally, users can re-time subtitles by adjusting the edges of these regions. Clicking on the waveform itself brings up a menu that offers options to add a subtitle at the clicked time or to reload the waveform, facilitating precise adjustments.
- **Waveform Controls** – Includes tools to zoom in and out of the waveform timeline and adjust the vertical scale of the waveforms. These controls are vital for detailed editing and navigating within the audio track, allowing users to fine-tune how they view and interact with audio and subtitle data.

This structured breakdown ensures that each component's function is clearly understood, promoting efficient interaction with the software and supporting the complex task of video and subtitle editing.

3.4 Tech Stack Selection

This section justifies the selected technologies and their suitability for developing a comprehensive tool for subtitle editing and video management, ensuring optimal performance and user experience across various platforms. The technologies chosen are shown in figure 3.5 below.

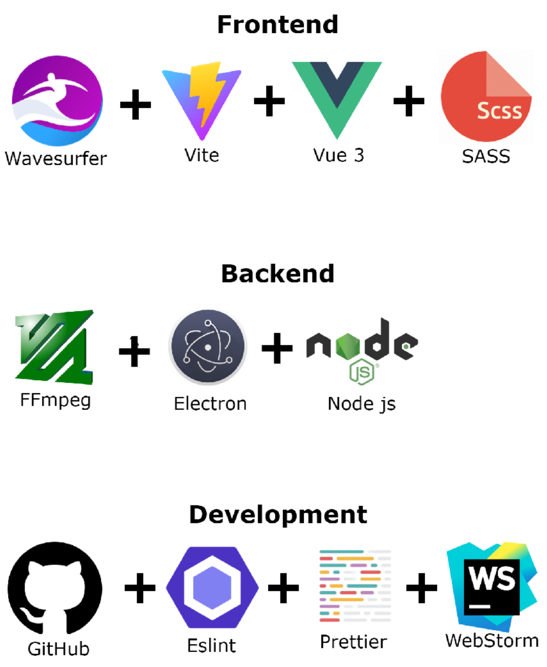


Figure 3.5: Technology stack

3.4.1 Frontend

- **Electron** – Electron 2.1.4 provides a framework for creating native applications using web technologies, which is ideal for this project as it allows the development of cross-platform desktop applications from a single codebase. This capability ensures that our application can operate seamlessly on Windows, macOS, and Linux, thereby broadening user accessibility and simplifying deployment.

- **Vue.js and Vite** – Vue.js is a progressive JavaScript framework designed for building user interfaces, and when combined with Vite, a modern front-end build tool, it enhances the application's responsiveness. This setup is particularly effective for managing dynamic UI elements like live subtitle editing and video playback controls, enabling quick updates and smooth interactions.

- **Wavesurfer.js** – As mentioned in 2.3.2, this customizable waveform audio visualization library allows for the visual representation of sound, essential for editing subtitles in sync with audio cues. Its ability to integrate directly with the user interface ensures a practical tool for precise audio track manipulations.

- **SASS** – As a powerful CSS preprocessor, SASS simplifies the management of complex stylesheets with features like variables, nested rules, and mixins, making the UI styling more maintainable and easier to develop.

3.4.2 Backend

- **Node.js** – Node.js serves as the backend framework, providing a robust environment for executing JavaScript code outside of a browser. This choice supports efficient handling of file operations and processing tasks which are crucial for video and subtitle manipulation.

- **FFmpeg** – FFmpeg 2.4.4 is indispensable for any video processing tool, offering comprehensive capabilities to record, convert, and stream audio and video. It plays a critical role in the application for processing videos and extracting audio tracks necessary for subtitle synchronization.
- **Subtitle** – A dedicated library for managing subtitle files, facilitating easy parsing, modifying, and writing of subtitles across various formats. This tool simplifies the core functionality of editing and aligning subtitles within the application.

3.4.3 Development

- **GitHub** – GitHub hosts the central repository for source code management and version control. It provides collaborative features essential for open-source development, allowing multiple developers to contribute effectively.
- **ESLint** – This tool aids in identifying and reporting on patterns found in ECMAScript/JavaScript code, making it possible to fix problems before they affect the application's functionality or cause developer conflicts.
- **Prettier** – Prettier is an opinionated code formatter that enforces a consistent style by parsing code and re-printing it with its own rules, thus enhancing the readability and maintainability of the source code.

3.4.4 Connecting frontend with backend

Communication between the frontend and backend is facilitated through Electron's IPC (Inter-Process Communication) mechanism as mentioned in 3. The `ipcMain` module in Electron's main process is used to handle incoming messages and commands from the renderer process (the frontend), which operates within the application's windows. Tasks that require more intensive processing or access to Node.js capabilities, such as file system operations or external command execution (e.g., using FFmpeg for video processing), are managed by the `ipcMain`.

On the frontend side, the `ipcRenderer` module is used within the Vue.js application to send messages and requests to the `ipcMain`. This allows the frontend to remain responsive and offload heavier tasks to the backend without freezing the user interface. It's an effective way to keep the user experience smooth, especially during resource-intensive operations.

3.4.5 Building and deployment

The building and deployment process of the application is meticulously structured to ensure an efficient and reliable setup. The process revolves around several key components that interact to create a robust system capable of handling both the user interface and the application's backend logic.

Firstly, the build process initiates with `Vite`, which compiles the `Vue.js` application. `Vite's` role is crucial as it bundles all the frontend assets efficiently, resulting in a highly optimized build. This build is output to the `dist` folder, which serves as the root for all compiled assets. The contents of this folder include the HTML, CSS, and JavaScript files that have been optimized and minified for production.

Electron's main script, typically named `index.js` or `main.js`, plays a pivotal role in bootstrapping the application. This script is responsible for creating and managing application windows and handling system-level interactions. It loads the contents of the dist folder to display the user interface, effectively turning the web application built with `Vue.js` into a native desktop application.

The final deployment of the application involves packaging the Electron application with all its dependencies into an executable file for Windows, macOS, or Linux. Tools such as Electron Builder or Electron Packager are typically used for this purpose, allowing developers to create a standalone application that users can install and run on their devices.

3.5 Application architecture design

This section sketches the foundational design of the application's architecture, illustrating the integration and functionality of individual processes within the application. This explanation includes the flow of data, the responsibilities of each component, and how they interact to maintain application state and functionality.

3.5.1 Application Processes Schema

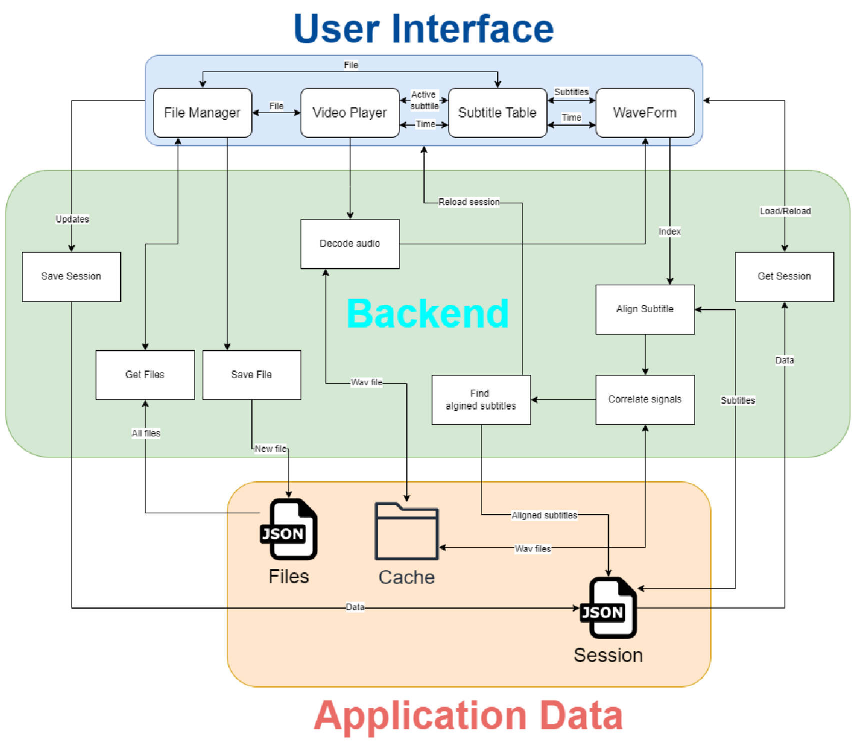


Figure 3.6: Application Schema

Figure 3.6 showcases the roles and interactions of application processes across three primary layers – User Interface, Backend, and Application Data Layer.

The application uses a `session.json` file to persistently store all data required by both the backend and the user interface. This JSON file is central to the application's ability to maintain its state across sessions. Data within this file is updated with each session change and is reloaded every time the application is launched. This mechanism not only preserves the state after the application is closed but also enables functionalities such as undo (Ctrl+Z) and redo (Ctrl+Y) by keeping track of historical states. Moreover, the user interface is designed to be dynamically updated by the backend via notifications to reload the session data when significant updates are made and stored in `session.json` .

The `FileManagement` component of the application leverages a `files.json` file to efficiently manage and cache data regarding all loaded video and subtitle files. This caching mechanism significantly speeds up the reopening of files by providing quick access through the *recent files* list within the `FileManager` component.

For handling audio, the application utilizes a dedicated cache directory to store `wav` files, which are generated from the video files using `ffmpeg` when a video is first opened in the `VideoPlayer` component. These audio files are then accessed by the `WaveForm` component, which relies on `wavesurfer.js` for rendering the audio waveforms visually.

The subtitle alignment process is initiated when the user selects a subtitle that acts as the reference for synchronization. The selected subtitle's corresponding audio segment, defined by its `start` and `end` times, is fetched from the cached `wav` file. This segment is then used by the backend to correlate with other audio signals to determine the optimal alignment. Lastly, the process compares all parsed audio segments from all of the subtitles to verify their alignment relative to the reference subtitle. Once alignment is confirmed, the adjusted subtitles are saved back into the `session.json` file. The frontend is subsequently notified to reload the session data to reflect these updates.

3.5.2 Application data synchronisation

In the frontend, several pieces of data need to be shared and updated globally among components to ensure proper reactivity and synchronization:

- **Video file** – This is used to reference the current video file across the `FileManager` and `VideoPlayer` components, enabling seamless file management and playback interaction.
- **Subtitle file** – Shared between the `FileManager` and `SubtitleTable`, this ensures that the currently loaded subtitles are accessible and can be displayed or edited as needed.
- **Playback time** – This critical piece of data is shared among `VideoPlayer`, `SubtitleTable`, and `WaveForm` to accurately align the video playback with the displayed subtitles and waveform visualization.
- **Subtitles** – The full list of subtitles is shared between the `SubtitleTable` and `WaveForm` to facilitate the display of subtitles on the waveform for easier editing and synchronization.
- **Active subtitle** – Shared between `SubtitleTable` and `VideoPlayer`, this data helps in highlighting the currently active subtitle within the video playback, enhancing the user's editing and viewing experience.

Chapter 4

Implementation

This chapter focuses on key and most challenging parts of implementation of the frontend and backend. Lastly it provides a analysis of project directory where files of frontend and backend are differentiated.

4.1 Frontend

This section breaks down key components and other modules of the user interface involving challenging parts of implementation. The final version of the user interface is slightly different than the designed version due to several factors. Throughout the development process, new ideas and feedback from initial user testing influenced modifications to the original design. Additionally, practical considerations such as improved usability, better workflow integration, and technical constraints led to further refinements. These changes were implemented to enhance the overall user experience, ensuring the application is both functional and intuitive.

4.1.1 Components and modules

The diagram shown in Figure 4.1 provides a comprehensive visualization of the various components and their dependencies within a the project, which was built using Vue.js. This graph was generated using Vue Developer Tools, a popular tool for inspecting and debugging Vue applications. Here's a detailed breakdown of the graph's elements and their interactions:

- **Vue Components** – Represented by green nodes, these are the primary building blocks of the Vue application. Each component defines its template, logic, and styling, encapsulating functionality in a modular way.
- **Javascript modules** – Shown in dark blue nodes, these files are probably used for utility functions, event handling, or service logic.
- **Styling Files** – Yellow and pink nodes represent styling files which define the visual aspect of the application. The SCSS files (`colors.scss`, `common.scss`) allow for use of variables, nested rules, and other features not available in plain CSS (`style.css`).
- **Utility and Framework Scripts** – Nodes like `vuetify.js` show the use of Vuetify, a Vue UI library with a collection of pre-made components that adhere to Material

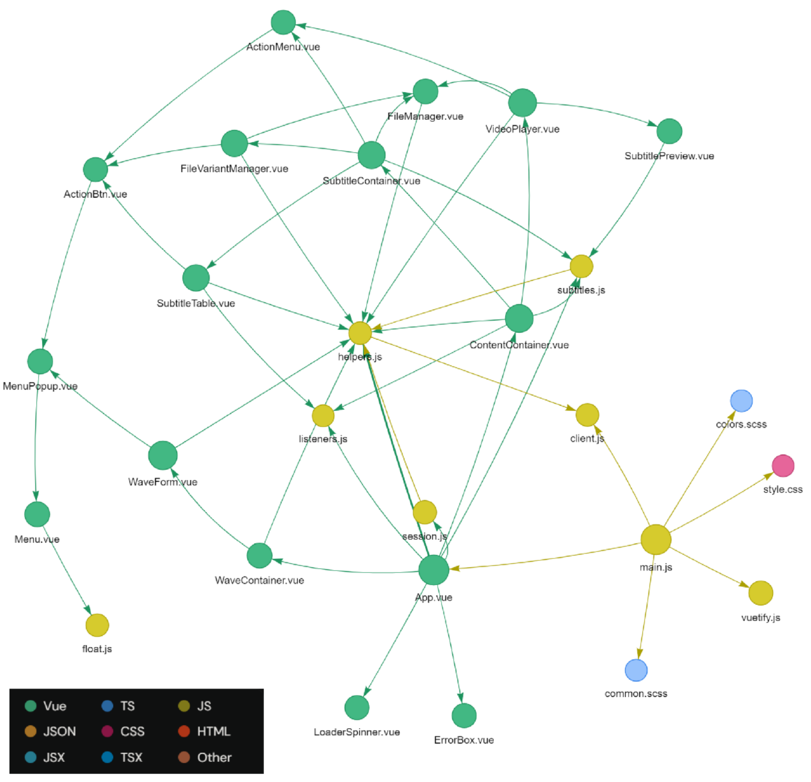


Figure 4.1: Modules's Graph

Design principles. This is indicated by a light blue node, which is linked to other components to provide consistent UI elements across the application.

- **Data and Session Management** – Shown with orange nodes, these are crucial for handling configuration and maintaining session state (`session.json`), essential for data persistence across user sessions.

The lines connecting these nodes illustrate the dependencies and data flow between components and scripts, indicating a well-structured frontend architecture where components are modular but interconnected, where `main.js` functions as the entry point initializing the Vue instance and other global settings and `App.vue` acts as the root component from which other components are derived. The arrows depict dependencies, starting from the parent node (where the arrow originates) and pointing to the child node (where it ends). This shows how components like `ContentContainer.vue` and `LoaderSpinner.vue` are integrated into `App.vue`. The widespread use of `helpers.js` across various components is indicated by multiple incoming arrows, which highlights its role in providing utility functions throughout most of the application.

4.1.2 Central Module Integration

The file `main.js` serves as the entry point for a Vue.js application, playing a crucial role in the initial setup and configuration of the entire application. This central script is responsible for several foundational tasks crucial for the robust functioning of the application.

The primary function of `main.js` is to create the root Vue instance. This involves importing the Vue library and other essential libraries that enhance Vue's capabilities, such as Vuex for state management or Vue Router for seamless navigation between views. It also binds the root component, `App.vue`, to an element in the HTML. This root component acts as the container for the entire application, with all other components nested within it.

Integrating various plugins and libraries into the application is also done in `main.js`. For instance, UI libraries like Vuetify are initialized here to ensure they are available throughout the application. This file is where global components are registered, making them accessible in any part of the application without the need for individual imports in each component.

```
import { createApp, reactive } from 'vue'
import './style.css'
import App from './App.vue'
import apiService from '@/utilities/client.js'
import vuetify from './plugins/vuetify'
import { PerfectScrollbarPlugin } from 'vue3-perfect-scrollbar'
import 'vue3-perfect-scrollbar/style.css'
import 'vuetify/dist/vuetify.min.css'
import '@mdi/font/css/materialdesignicons.min.css'
import './css/colors.scss'
import './css/common.scss'

const app = createApp(App)
const globalError = reactive({ message: '' })
const globalLoad = reactive({ message: '' })
const globalUpdate = reactive({ targets: [], stuck: false })

// Use Vuetify instance here
app.use(vuetify)
app.use(PerfectScrollbarPlugin)

app.config.globalProperties.$apiService = apiService
app.config.globalProperties.$error = globalError
app.config.globalProperties.$loading = globalLoad
app.config.globalProperties.$update = globalUpdate

app.mount('#app')
```

Listing 4.1.1: Central module - `main.js`

The script shown in listing 4.1.1 begins by importing the necessary modules and CSS for the Vue application. The code sets up global reactive states `globalError`, `globalLoad`, and `globalUpdate`, which are used for managing errors, loading states, and updates that cannot use the usual reactivity system due to performance concerns. These states are then assigned to Vue's global properties to be accessible throughout the application. Variable `globalError` is watched to display error pop-ups across the application, which helps in centralizing error management and displaying messages whenever an error occurs in any part of the application. Variable `globalLoad` is monitored to show loading overlays with progress circles. It is particularly useful in user interface feedback for operations that take a significant amount of time. Last global variable `globalUpdate` is a stack-like structure for updating parts of the application that cannot utilize Vue's reactivity system directly due to performance reasons. Instead, specific functions within certain components watch this state and make updates accordingly. Last line `app.mount('#app')` ensures that the application is mounted to the DOM element with the `ID 'app'`.

4.1.3 Session module

This module serves as a comprehensive session management system within the application, particularly designed to handle dynamic user interactions that frequently modify session data. This module ensures the persistence and reversible modification of session data, which is crucial due to editing and real-time content manipulation.

Module Operation

The module initializes by setting up a reactive session object using Vue's `ref`, which holds the session's current state including identifiers, timestamps, media file details, and historical data for undo and redo actions. This initial setup is populated with blank data which will later be replaced upon loading actual session data from the backend.

Session Loading and Saving

Upon component mounting, indicated by the `onMounted` hook, the module calls the `load` function to fetch existing session data from the backend. This function attempts to retrieve a session JSON file and populates the session state with its contents. If no existing data is found, a new session with a unique ID and timestamp is initialized and immediately saved using `$apiService.sendMessage('save-session')` to ensure that a valid session state is always present.

Monitoring Changes and Saving Session Data

The module uses Vue's `watch` to monitor changes to `session.value`. Whenever a change occurs, the module compares the new session data against the last session snapshot stored in `lastSession`. If differences are found, and they are not due to keys listed in `keyBlacklist` (which are ignored to prevent excessive saves), the new state is serialized and pushed into the appropriate history stack (undo or redo) and saved back to the backend. This mechanism ensures that every significant change is captured and can be reversed if necessary.

Undo and Redo Functionality

The undo and redo mechanisms are fundamental features of this module. They allow the user to navigate through their change history, either reverting to a previous state or reapplying changes they have rolled back. When an undo or redo operation is executed, it triggers a notification to update specific parts of the application by pushing 'undo-redo' into `$update.targets`. This informs other components that a major data change has occurred, allowing the application to react and update accordingly.

Session data

```
{
  "data": [
    {
      "id": "",
      "sync": true,
      "videoFile": "C:/Users/user/example.mkv",
      "subtitleFile": "C:/Users/user/example.srt",
      "subtitleRows": [],
      "offset": 43264.000000000124,
      "offsetMs": 43264.000000000124
    },
    {}
  ],
  "history": {
    "undoList": [],
    "redoList": []
  }
}
```

Listing 4.1.2: Session data structure

Breaking down the session data structure in listing 4.1.2 can provide insight into the data that the application uses and what application logic is build for it:

1. **Data** – This array holds objects each representing a media content container. Each container includes:
   - `id` – A unique identifier for each content set, useful for tracking and managing multiple files.
   - `sync` – A boolean indicating whether the subtitle has been successfully synchronized with the video.
   - `videoFile` and `subtitleFile` – Paths to the video and subtitle files, respectively.
   - `subtitleRows` – An array of subtitle entries, each containing timing and text data.
   - `offset` and `offsetMs` – The amount of time (in milliseconds) by which subtitles have been shifted to achieve synchronization.
2. **History** – Manages the undo and redo functionalities:
   - `undoList` and `redoList` – Arrays that store historical states of the application data, enabling users to revert or reapply changes made during the editing process.

It is important to note that not all data used by the application is saved in this structure due to too many frequent updates like for example the current playback time of the video.

4.1.4 App

The App component serves as the main hub of the application, acting as the parent for all other components. It is essential for integrating various parts of the system, such as `ContentContainers` and `WaveContainer`, creating a cohesive user interface. This layout is effectively illustrated in figure 4.2 which showcases how the App component organizes the view into distinct sections for content management and waveform display.

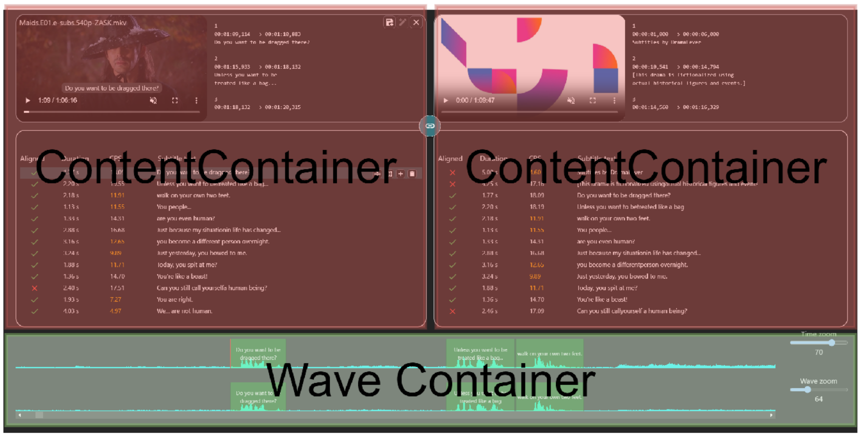


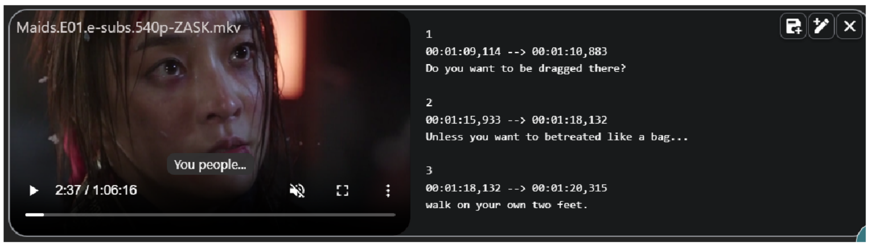
Figure 4.2: Application base layout

Handling Updates

Within the App component, there is a sophisticated mechanism for handling updates across the application, particularly those related to subtitle management. These updates are intricately linked with session management functionalities like undo, redo, and load operations managed by `session.js`. Subtitle updates, due to their complexity, are often batched and pushed to `$update.targets`. This system allows for handling multiple updates at once, ensuring that all components remain in sync. Detailed handling of these updates can be referenced in 4.1.7. If concurrent editing features are enabled, the component ensures that updates affecting one subtitle are also applied to other synchronized subtitles, maintaining consistency across the platform.

4.1.5 Video Player

This component is designed to manage and display video content along with its associated subtitles. It renders the video on the right side of the interface and features a scrollbar for previewing subtitles in SRT format, showing how they will appear once saved as can be seen in figure 4.3. The video player inherits standard controls from the HTML `<video>` element, which include play, pause, seek, and volume control, allowing users to interact with the video directly within the UI.



```
1
00:01:09,114 --> 00:01:10,883
Do you want to be dragged there?

2
00:01:15,933 --> 00:01:18,132
Unless you want to betreated like a bag...

3
00:01:18,132 --> 00:01:20,315
walk on your own two feet.
```

Figure 4.3: Video Player

In addition to video playback, the component supports file management operations through embedded controls. Users can open a new video file, embed subtitles directly into the video, or close the current video file. These functions are facilitated through an

ActionMenu component that dynamically updates its items based on the state of the video file and the presence of subtitles. For instance, the save action becomes available only when both a video file and subtitles are loaded and the subtitles are ready to be embedded. This ensures that users only see relevant options, streamlining the interface and preventing errors. Additionally, the edit action allows users to change the current video file if one is already loaded, or open a new file if none is present. Similarly, the close action is enabled when a video file is open, providing a quick way to clear the current session.

Upon video play, the component emits an update event update:time to synchronize the current time across all components within the application that require this data. This ensures that all related components are aware of the current video time.

Currently playing subtitle

The currently playing subtitle is displayed in an overlay style directly on the video, which dynamically adjusts based on user interaction and video playback status. The styling and animation are carefully designed to enhance readability and user experience. Listing 4.1.3 contains the CSS and the conditions under which the style changes.

```
<style scoped>
.subtitle-overlay {
  position: absolute;
  bottom: 4rem;
  z-index: 99;
  width: 100%;
  transition: transform ease 0.2s;
}
.subtitle-overlay-content {
  color: white;
  background: rgba(50, 50, 50, 0.9);
  margin: auto;
  padding: 0.1rem 0.5rem;
  border-radius: 0.5rem;
  font-size: 0.85rem;
}
</style>
```

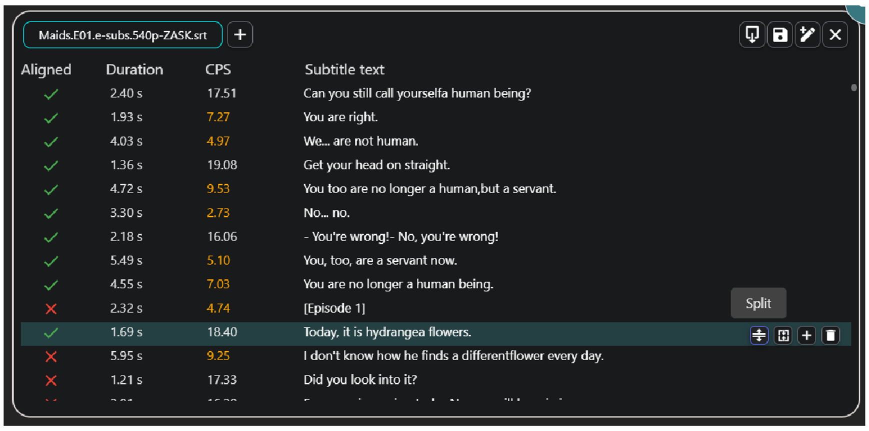
Listing 4.1.3: Subtitle overlay styling

The .subtitle-overlay class in listing 4.1.3 defines the positioning and basic appearance of the subtitle overlay, ensuring it is placed at a consistent location above the video controls. The overlay content is styled to ensure the text is legible against potentially varying video backgrounds. The transition for the transform property is defined to smooth out the movement of the subtitle text, which translates vertically based on whether the video controls are being interacted with.

4.1.6 Subtitle Table

The SubtitleTable component, as visualized, provides a user-friendly layout designed for managing subtitle files effectively. It features a structured table format that displays various attributes of subtitle entries, enabling easy editing and manipulation of subtitle data. It is perhaps the most important component, because user will be editing subtitles here.

Figure 4.4 showcases how the SubtitleTable component looks like inside the application.


Maids.E01.e-subs.540p-ZASK.srt

Aligned	Duration	CPS	Subtitle text
✓	2.40 s	17.51	Can you still call yourselfa human being?
✓	1.93 s	7.27	You are right.
✓	4.03 s	4.97	We... are not human.
✓	1.36 s	19.08	Get your head on straight.
✓	4.72 s	9.53	You too are no longer a human,but a servant.
✓	3.30 s	2.73	No... no.
✓	2.18 s	16.06	- You're wrong!- No, you're wrong!
✓	5.49 s	5.10	You, too, are a servant now.
✓	4.55 s	7.03	You are no longer a human being.
✗	2.32 s	4.74	[Episode 1]
✓	1.69 s	18.40	Today, it is hydrangea flowers.
✗	5.95 s	9.25	I don't know how he finds a differentflower every day.
✗	1.21 s	17.33	Did you look into it?

Figure 4.4: Subtitle Table

Component Layout

The header of the table is laid out horizontally and includes columns such as `Aligned`, `Duration`, `CPS` (Characters Per Second), and `Subtitle text`. Each header column is dynamically set to display a title unless it's reserved for action buttons, offering a clear indicator of the data below it. Below the headers, the body of the table is wrapped in a `perfect-scrollbar` component, ensuring smooth scrolling even with a large number of subtitle entries. This scrollable area contains rows, each corresponding to a subtitle segment.

Each row presents individual subtitle data, aligning with the headers above. Special icons are used to indicate whether a subtitle is properly aligned (green check for yes, red cross for no). `Duration` and `CPS` are calculated and displayed dynamically, with CPS values colorized based on certain criteria to provide visual feedback about subtitle pacing. The `Subtitle text` field in each row is editable, allowing for direct modifications. This input is styled to take up the full width of its column when the header is `Subtitle text`.

Active Subtitle

The active subtitle is highlighted in blue, making it visually distinct in the interface. It is automatically selected when a user interacts with a corresponding section on the `Waveform` component, as detailed in 4.1.7. When aligned and concurrent editing is enabled, this active status is synchronized across other subtitle tables.

For the active subtitle, specific action buttons are presented under the `actions` header. These buttons facilitate operations such as splitting the subtitle, merging it with others, inserting a new subtitle immediately after, and deleting it. Equipped with tooltips, each button provides clear guidance and is designed to perform its respective action upon being clicked. This setup is essential for efficient subtitle editing, allowing direct manipulation of subtitles through the table interface, as further supported by keyboard shortcuts described in 4.1.6.

Keyboard Shortcuts

The component also supports various keyboard shortcuts to enhance productivity and ease of use. The `handleMultipleKeyCombinations` function is designed to respond to specific

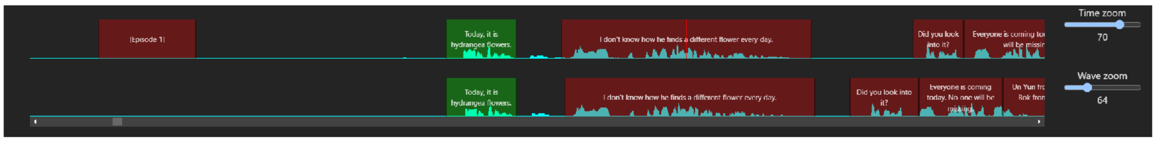

|Episode 1|
Today, it is hydrangea flowers.
I don't know how he finds a different flower every day.
Did you look into it?
Everyone is coming to
will be missin
Today, it is hydrangea flowers.
I don't know how he finds a different flower every day.
Did you look into it?
Everyone is coming today. No one will be missing.
Un Yun fr
Bok fron

Figure 4.5: Wave Container

keyboard inputs, providing shortcuts for common actions. For instance, `arrow` keys are used to navigate between subtitles, with additional functionality when combined with the `Ctrl` key to jump to the beginning or end of the list. Other shortcuts include the `Q` key to play from the active subtitle, `Enter` to select a subtitle, and `D` along with Shift to duplicate a subtitle. Splitting a subtitle can be achieved by pressing `S` with Shift, and pressing `Insert` adds a new subtitle after the active one.

4.1.7 Wave Form

The `WaveForm` component is a crucial display tool for managing audio and subtitle synchronization within this video editing software. It utilizes the `wavesurfer.js` library, as introduced in 2.3.2, to display audio waveforms from video files. This component also employs the regions plugin from `wavesurfer.js` to visualize subtitles, enabling users to interact directly with the audio timeline to re-time, add, or delete subtitle entries. The waveforms of multiple videos are shown simultaneously, providing a comprehensive overview of the audio and subtitle synchronization, as illustrated in figure 4.5. They are all wrapped in a component called `WaveContainer`, which handles concurrent scrolling, time zoom and wave zoom. Concurrent scrolling is later broken down in 4.1.7.

Rendering and performance

When it comes to rendering, the `WaveForm` displays a progress circle during processing to indicate that data is being loaded or adjusted. The waves of the waveform are generated into the `wavesurfer.js's` canvas from audio data extracted using `ffmpeg` on the backend and are stored in a cache directory. This cached data is accessed via a URL path, allowing for efficient loading and rendering of the waveforms in the user interface.

As mentioned in 4.1.2, there is a `globalUpdate` variable, which is used for parts of the application that cannot utilize Vue's reactivity system directly due to performance reasons. One of these parts is rendering of the subtitles on the `WaveForm`, so this variable is watched as shown in listing 4.1.4.

```
const handleSubtitleUpdates = () => {
  const { name, target, idx } = $update.targets[$update.targets.length - 1]
  if (idx !== props.idx) return
  let region = null
  if (target && target.id && target.id !== -1)
    region = wsRegions.value.regions.find(region => region && region.subId === target.id)
  switch (name) {
    case 'subtitles':
      updateRegionsFromSubtitles()
      break
    case 'subtitle-update':
      updateRegion(target, region)
      break
    case 'subtitle-delete':
      if (region) {
        region.remove()
        delete wsRegions.value.regions[wsRegions.value.regions.indexOf(region)]
      }
      break
    case 'subtitle-add':
      addRegion(target)
      break
    default:
      break
  }
  $update.targets.pop()
}
watch(
  () => $update.targets,
  () => {
    if (!$update.targets.length) return
    if ($update.targets[$update.targets.length - 1] === 'undo-redo') {
      updateRegionsFromSubtitles()
      if (props.idx === Array.from(document.getElementsByClassName('waveform')).length - 1)
        $update.targets = []
    } else {
      handleSubtitleUpdates()
    }
  },
  { deep: true },
)
```

Listing 4.1.4: Handling subtitle updates

The `handleSubtitleUpdates` function in listing 4.1.4 is triggered by a watcher on the `$update.targets` array, which stores details about which subtitles need updating. This function specifically checks the last item in the `$update.targets` array to determine the nature of the update required:

- **Update single subtitle** – If a single subtitle's details have changed, method `updateRegion` is called to update just that specific region (or subtitle segment) on the waveform. This is more efficient than redrawing all subtitles because it only addresses the changed element.

- **Update all Subtitles** – Method `updateRegionsFromSubtitles` re-renders all subtitle regions on the waveform. It's used when a more comprehensive update is necessary, such as after undoing or redoing changes, which might affect multiple subtitles.

- **Delete subtitle** – If a subtitle is to be removed, the corresponding region is identified and deleted from the waveform display. This also involves cleaning up the internal array that tracks these regions, ensuring that the application's state remains accurate and up-to-date.

- **Add new subtitle** – When a new subtitle is added, this function creates a new region on the waveform to represent it visually.

Interactions with subtitles

Interactivity within the `WaveForm` includes a variety of menu options that enhance user experience. Clicking on a subtitle within the waveform opens a popup menu that allows the user to align or clear alignment and delete the subtitle. If a click occurs on the waveform where no subtitle exists, it opens a popup menu that to add a subtitle, reload the waveform, and clear any existing alignments are presented. These features facilitate precise control over subtitle timing and placement directly from the waveform interface.

Alignment

The alignment process within the `WaveForms` is designed to ensure accurate synchronization of audio tracks with subtitles. When a user aligns a subtitle, the `start` and `end` times of the referenced subtitle(which was clicked on) are sent to the backend, which then updates the session data with the aligned subtitles and calculates any necessary offsets. This offset is applied as a left margin to the start of one of the `WaveForms`, helping to visually align the audio signals beneath the subtitles.

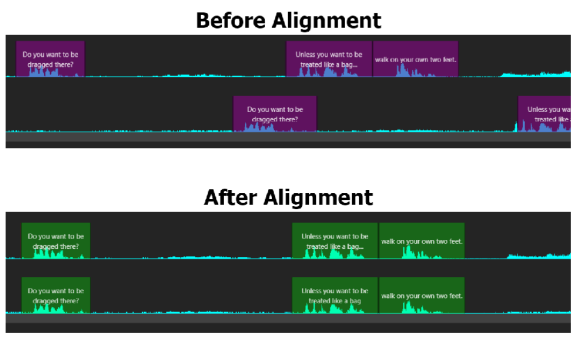

Before Alignment
Do you want to be dragged there?
Unless you want to be treated like a bag...
walk on your own two feet.
Do you want to be dragged there?
Unless you wa treated like
After Alignment
Do you want to be dragged there?
Unless you want to be treated like a bag...
walk on your own two feet.
Do you want to be dragged there?
Unless you want to be treated like a bag
walk on your own two feet.

Figure 4.6: Subtitle alignment

This alignment is key to ensuring that the audio and subtitles are in sync, which can be visually confirmed as from figure 4.6.

Re-timing

The re-timing of subtitles within the `WaveForm` component allows users to interactively adjust the timing of subtitles directly through the waveform interface. This interaction is done in two primary ways:

1. **Moving subtitle on the timeline** – Users can click, hold, and drag a subtitle region left or right to shift its start and end times simultaneously. This method maintains the duration of the subtitle but changes its positioning within the timeline, allowing for quick adjustments in sync with audio cues.

2. **Modifying duration by adjusting edges** – By holding and dragging the right or left edge of a subtitle region, users can change either the start time or end time, respectively. This action affects the duration of the subtitle, making it longer or shorter depending on the direction of the drag.

To ensure subtitles do not overlap, which can confuse viewers and disrupt the viewing experience, the `WaveForm` component includes specialized listeners within the `wavesurfer.js` framework. The provided code snippet in listing 4.1.5 is crucial for managing these interactions.

```
wsRegions.value.on('region-created', region => {
  region.content.style.margin = 'auto'
  region.listeners.update.add(() => handleRegionDrag(region))
})
```

Listing 4.1.5: Region listeners

When a region (subtitle) is created, a margin is automatically set, and an update listener is added. This listener invokes `handleRegionDrag` as shown in listing 4.1.6 whenever the region is manipulated.

```
const handleRegionDrag = region => {
  const regions = wsRegions.value.getRegions().sort((a, b) => a.start - b.start)
  const regionIndex = regions.indexOf(region)
  const regionStart = region.start
  const regionEnd = region.end

  // check if region is overlapping with his left neighbor
  const nearestLeftRegion = regions
    .filter((r, i) => i < regionIndex)
    .reduce((acc, r) => (r.end > acc.end ? r : acc), { end: 0 })
  if (nearestLeftRegion.end > regionStart) {
    region.start = nearestLeftRegion.end
    region.end = regionEnd + (nearestLeftRegion.end - regionStart)
  }

  // check if region is overlapping with his right neighbor
  const nearestRightRegion = regions
    .filter((r, i) => i > regionIndex)
    .reduce((acc, r) => (r.start < acc.start ? r : acc), { start: 1000000 })
  if (nearestRightRegion.start < regionEnd) {
    region.end = nearestRightRegion.start
    region.start = regionStart - (regionEnd - nearestRightRegion.start)
  }
}
```

Listing 4.1.6: Handling region updates

Function shown in listing 4.1.6 retrieves all existing subtitle regions and sorts them by their start times. Each region is then indexed to identify its position relative to others. Handling drag operations for each moved region is done by checking if there is overlap with the subtitle directly to the left. If the current region's start time intrudes into the left neighbor's end time, both the start and end times of the current region are adjusted forward to eliminate the overlap. Similarly, if the current region's end time extends into the start time of the right neighbor, it's pulled back to prevent overlap.

Scrolling into playback

This application behaviour is important in order to ensure that the `WaveContainer` display scrolls synchronised with video playback, providing a smooth auto-scrolling experience as the video progresses. It is done by dynamically calculating the appropriate scroll position within the waveform container based on the current playback time of the video as can be seen in listing 4.1.7.

```
const handleTimeUpdate = () => {
  if (ws.value) {
    ws.value.setTime(time.value)
  }
  const container = document.getElementById('waveform-container')
  const currentContainerWidth = ws.value.renderer.container.getBoundingClientRect().width
  const maxWidth = Math.max(
    ...Array.from(document.getElementsByClassName('waveform')).map(
      x => x.children[0].shadowRoot.querySelector('.scroll').getBoundingClientRect().width,
    ),
  )
  const containerRatio = currentContainerWidth / maxWidth
  const cursorPosPercent = ws.value.getCurrentTime() / ws.value.getDuration()
  const scrollPos = container.scrollWidth * cursorPosPercent * containerRatio - 100
  container.scrollTo({
    left: scrollPos,
  })
}
```

Listing 4.1.7: Handling time updates

The method `handleTimeUpdate` in listing 4.1.7 then computes the ratio of the container's width to this maximum width (`containerRatio`), ensuring that the scroll position scales correctly relative to the size of the waveform displayed. The position of the playback cursor within the `WaveContainer`'s (`cursorPosPercent`) is calculated by dividing the current playback time by the total duration of the video, resulting in a percentage that represents how far along the video has played.

The desired scroll position (`scrollPos`) is then calculated by multiplying the total scrollable width of the container by both the cursor's percentage position and the container ratio. A fixed offset of 100 pixels is subtracted to align the scroll position accurately with the cursor within the `WaveContainer`'s display, ensuring the cursor is ideally positioned within the viewport.

Finally, the `WaveContainer` is instructed to scroll to this calculated position using the `scrollTo` method, with smooth scrolling enabled.

Concurrent Editing

The concurrent editing functionality enhances the flexibility of subtitle editing. A central button within the App toggles a popup menu that offers several options depending on whether a subtitle is currently aligned. If no subtitle is aligned, the first action is to align the active subtitle from the `SubtitleTable`. From this menu, users can choose to re-time subtitles, edit their text concurrently, delete aligned subtitles, or add new subtitles within aligned sections.

Re-timing is performed directly on the `WaveForm`, similar to the process described in 4.1.7, but it also adjusts the timing of any subtitles that are aligned with the currently active one. Concurrent text editing occurs within the `SubtitleTable` (as detailed in 4.1.6), allowing for simultaneous modifications to the text of aligned subtitles if this feature is enabled.

Popups

Within the application, the management of user notifications and loading states is handled through specialized popups. These include the `ErrorBox` for displaying error messages and the `LoaderSpinner` for indicating loading states, enhancing user feedback and application responsiveness.

```
watch(errorMessage, value => {
  if (value) {
    setTimeout(() => {
      $error.message = ''
      errorMessage.value = ''
    }, 5000)
  }
})
```

Listing 4.1.8: Handling error message display

Command shown in listing 4.1.8 attaches a `watch` function to the `errorMessage` reactive property. When an error message is detected (i.e., `value` becomes true), a timer is initiated using `setTimeout`. This timer clears the error message stored in both `errorMessage` and the global `$error.message` after 5 seconds, ensuring that error notifications are visible just long enough to inform the user without lingering on the screen.

```
watch($error, value => {
  if (value) {
    errorMessage.value = value.message
  }
})
```

Listing 4.1.9: Updating error message from global error state

Command shown in listing 4.1.9 listens for changes in the `$error` reactive object. If there is a change, the `errorMessage` reactive property is updated to reflect the new message. This ensures that any errors affecting the application are quickly displayed in the `ErrorBox`, providing timely feedback to the user.

```
watch(
  $loading,
  value => {
    if (value) {
      loadingMessage.value = value
    }
  },
  { immediate: true },
)
```

Listing 4.1.10: Handling loading state display

Command shown in listing 4.1.10 watches the `$loading` reactive state with the `immediate: true` option, which triggers the function as soon as the component is initialized. This watch captures and updates the `loadingMessage` reactive property whenever `$loading` changes, signaling the start or end of a loading process. The `LoaderSpinner` is thus controlled to show or hide based on the current loading status, keeping the user informed about the application's activity. These reactive watch functions effectively manage the dynamic display of errors and loading states within the application, providing critical feedback to users in a controlled and timely manner. This setup ensures that the user interface remains clear, informative, and responsive to the ongoing activities within the application.

4.2 Backend

This section delves into the backend processes essential for handling various multimedia tasks such as parsing subtitle files, converting audio formats, aligning audio signals, and synchronizing subtitles with audio. These functions are critical for ensuring that the media content is properly processed and synchronized as fast as possible for an optimal user experience.

4.2.1 Processing Subtitles

The handling of subtitle files in this application involves specific functions that utilize the `FFmpeg` tool and `subtitle` library to manage subtitles directly from video files. These functions are tailored to extract subtitles, merge them back into video files, save subtitles in different formats, and parse subtitle data effectively.

Extracting subtitles from video

```
const extractSubtitles = inputFilePath => {
  return new Promise((resolve, reject) => {
    inputFilePath = inputFilePath
      .replace(/\\/g, '/')
      .replace(/\//g, '\\')
      .replace(/file:\\\\\\/g, '')
      .replace(/\r\n/, '')
    if (!fs.existsSync(inputFilePath)) {
      console.error('The input file does not exist:', inputFilePath)
      reject(new Error('Input file not found'))
      return
    }
    const outputFilePath = `${process.cwd()}\\videos\\
    ${path.basename(inputFilePath, path.extname(inputFilePath))}.srt`
    ffmpeg(inputFilePath)
      .outputOptions([
        '-map 0:s:0', // Assumes the first subtitle stream; adjust if needed
        '-c:s srt', // Specifies copying the subtitle stream as-is, assuming it's in SRT format
      ])
      .save(outputFilePath)
      .on('end', () => {
        console.log(`Subtitles have been extracted to: ${outputFilePath}`)
        // Read and parse the subtitle file
        fs.readFile(outputFilePath, 'utf8', (err, data) => {
          if (err) {
            console.error('Error reading the subtitle file: ' + err.message)
            reject(err)
          } else {
            try {
              const subtitlesObj = parseSync(data)
              resolve(subtitlesObj)
            } catch (parseError) {
              console.error('Error parsing the subtitle file: ' + parseError.message)
              reject(parseError)
            }
          }
        })
      })
      .on('error', err => {
        console.error('An error occurred: ' + err.message)
        reject(err)
      })
  })
}
```

Listing 4.2.1: Extract subtitles from video

Function shown in listing 4.2.1 uses FFmpeg to extract SRT subtitle streams from video files. It targets the embedded subtitle tracks within the video file, extracting them into standalone SRT files. This capability is particularly useful for workflows that require subtitles to be edited or processed separately from their corresponding video. The function modifies the path to accommodate file system differences, checks the file's existence, and then executes the extraction, saving the subtitles to a specified output path.

Embedding subtitles to video

```
const mergeSubtitles = (inputFilePath, subtitles, outputFilePath) => {
  return new Promise((resolve, reject) => {
    ffmpeg(inputFilePath)
      .input(subtitles)
      .outputOptions(['-c copy', '-c:s mov_text'])
      .save(outputFilePath)
      .on('end', () => {
        console.log(`Subtitles have been merged to: ${outputFilePath}`)
        resolve(outputFilePath)
      })
      .on('error', err => {
        console.error('An error occurred: ' + err.message)
        reject(err)
      })
  })
}
```

Listing 4.2.2: Embed subtitles into video

Function shown in listing 4.2.2 integrates the SRT files back into the original video using FFmpeg. After subtitles have been modified or created, they need to be embedded back into the video. This function allows for the inclusion of new or edited subtitle tracks without altering the original audio and video streams, ensuring that the final product is ready for playback with the updated subtitles seamlessly integrated.

Loading and saving subtitle files

```
const loadSubtitles = subtitles => {
  return parseSync(subtitles)
}

const saveSubtitles = (subtitles, outputFilePath, preview = false) => {
  const format = outputFilePath.endsWith('.vtt') ? 'WebVTT' : 'SRT'
  const srtString = stringifySync(subtitles, { format })
  if (preview) return srtString
  fs.writeFileSync(outputFilePath, srtString)
}
```

Listing 4.2.3: Load and save subtitles

Function shown in listing 4.2.3 uses the subtitle library to load and save subtitle files. The `loadSubtitles` function is crucial for loading subtitles from files and converting them into a structured format that can be easily accessed and manipulated within the application. The `saveSubtitles` function converts subtitle data into a textual format like SRT or WebVTT and writes it to a file. The subtitle library provides functions to serialize subtitle objects according to the specifications of different subtitle formats.

4.2.2 Audio Conversion

Audio conversion is a critical step in preparing media for efficient processing and visualization within this application, particularly when dealing with complex and long audio-visual content. Therefore, a decision to convert videos into `WAV` files was made, due to relatively long waveform loading from video into waveform using the `wavesurfer.js` library. Currently `wavesurfer.js` uses a local path to fetch the decoded audio data from cached `WAV` file. These cached files are also useful when aligning audio signals, which is the subject of section 4.2.3.

```
const convertToWave = (videoFilePath, wavFilePath, sampleRate) => {
  return new Promise((resolve, reject) => {
    ffmpeg(videoFilePath)
      .audioCodec('pcm_s16le') // Set the audio codec
      .audioFrequency(sampleRate) // Set the sample rate
      .audioChannels(1) // Set to mono audio
      .format('wav') // Set the output format
      .on('end', () => {
        console.log(`Audio extracted to: ${wavFilePath}`)
        resolve(wavFilePath)
      })
      .on('error', err => {
        console.error('An error occurred: ' + err.message)
        reject(err)
      })
      .save(wavFilePath) // Output file path
  })
}
```

Listing 4.2.4: Convert video audio to WAV file

Function shown in listing 4.2.4 sets the audio codec to `pcm_s16le`, which is a format for raw audio that strikes a balance between file size and audio quality, ideal for processing and quick loading. The audio is converted to a mono track to reduce complexity and size, and the sample rate is explicitly set to `8000 Hz`. This lower sample rate is sufficient for visual waveform analysis while significantly reducing the data load, which accelerates the waveform rendering on the frontend. The output `WAV` file is saved into a cache directory, ensuring that it is readily accessible for subsequent operations or multiple accesses, which enhances the application's responsiveness and user experience.

4.2.3 Audio signal correlation

Audio signal correlation is a critical technique used in audio processing to determine how closely one audio signal matches a segment of another audio signal.

The objective is to compute a correlation score that quantifies the similarity between two audio signals. This involves analyzing the audio data to extract meaningful features and then comparing these features to find the best alignment or match. One common approach is to use Mel-Frequency Cepstral Coefficients (MFCCs), which are capable of capturing the key characteristics of an audio signal. The correlation between these coefficients can then be used to determine how similar two audio segments are.

4.2.4 Extraction of MFCCs

```
const extractMFCCs = (audioData, sampleRate, bufferSize) => {
  Meyda.sampleRate = sampleRate
  Meyda.bufferSize = bufferSize // A common bufferSize for MFCC
  const mfccs = []

  for (let i = 0; i < audioData.length; i += Meyda.bufferSize) {
    let buffer = audioData.slice(i, i + Meyda.bufferSize)
    if (buffer.length === Meyda.bufferSize) {
      const mfcc = Meyda.extract('mfcc', buffer)
      mfccs.push(mfcc)
    }
  }

  return mfccs
}
```

Listing 4.2.5: Extraction of MFCCs

The function `extractMFCCs` in lising 4.2.5 processes audio data to compute a series of MFCCs by using the `Meyda.js` library, where sampleRate of `8000 Hz` is used just like in audio conversion 4.2.2, because it uses the same audio data or segments from it as are saved in the `WAV` file. Below is a mathematical overview of how exactly are the MFCCs computed:

Let *s[n]* be the input audio signal, where n is the sample index. The MFCC extraction process involves dividing the audio signal into overlapping frames and applying a window function. For each frame, the Fourier transform is calculated, followed by the Mel filter bank processing, and finally the discrete cosine transform to obtain the MFCCs:

$$MFCC = DCT\left(\log\left(MelFilter\left(|FFT(window(s[n]))|^2\right)\right)\right) \tag{4.1}$$

In equation 4.1:

- DCT denotes the Discrete Cosine Transform
- *MelFilter* represents the Mel scale filter bank application
- FFT is the Fast Fourier Transform
- *window* refers to the applied window function

4.2.5 Cross-Correlation of MFCCs

```
const crossCorrelateMFCC = (mfccSegment, mfccSignal) => {
  let maxCorr = -Infinity // Start with a very low correlation score.
  let bestOffset = 0

  for (let offset = 0; offset <= mfccSignal.length - mfccSegment.length; offset++) {
    let sum = 0

    for (let i = 0; i < mfccSegment.length; i++) {
      let diffSum = 0
      for (let j = 0; j < mfccSegment[i].length; j++) {
        // Calculate squared difference for each MFCC coefficient
        let diff = mfccSegment[i][j] - mfccSignal[i + offset][j]
        diffSum += diff * diff
      }
      sum -= diffSum // Subtract to simulate correlation (minimizing distance)
    }

    // We look for the maximum since sum is negative (maximally less negative is better)
    if (sum > maxCorr) {
      maxCorr = sum
      bestOffset = offset
    }
  }

  return { bestOffset, maxCorr: -maxCorr }
}
```

Listing 4.2.6: Cross-Correlation of MFCCs

The function `crossCorrelateMFCC` shown in listing 4.2.6 is used to find the best alignment between two sets of MFCCs. The cross-correlation is performed by shifting the segment MFCC array over the signal MFCC array and calculating the sum of squared differences for each potential alignment. The alignment with the minimum squared difference (maximally negative sum in this implementation) indicates the best match:

$$\mathrm{Corr}(m, s) = \max_k \left(- \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (m[i][j] - s[i+k][j])^2 \right) \tag{4.2}$$

In equation 4.2:

- m is the MFCC array of the segment,
- s is the MFCC array of the signal,
- N is the number of frames in the segment,
- M is the number of coefficients per frame,
- k is the offset in the signal where the segment is being compared.

The resulting offset from function `crossCorrelateMFCC` is later used for positioning the subtitle in the correct part of the audio signal or checking if the subtitle is aligned, which is the objective of next subsection.

4.2.6 Subtitle Alignment

This section outlines the methodology for aligning subtitles based on the correlation of their audio signals, focusing primarily on backend processes. The alignment procedure begins by identifying and synchronizing a specific subtitle from the frontend interface, where the start and end times of the subtitle are relayed to the backend.

A complete process is showcased in figure 4.7, which consits of aligning segment to all audio and aligning rest of the subtitles.

Initial alignment

The first step in the alignment process involves positioning a selected subtitle's audio segment within the complete audio track of the video to which the subtitle is being aligned. The backend performs the following operations:

- Audio files pertinent to the alignment are loaded into memory to facilitate rapid access and processing.
- Mel-Frequency Cepstral Coefficients (MFCCs) are extracted from both the subtitle's segment and the entire audio track.
- The extracted MFCCs are then cross-correlated to determine the best offset, ensuring that the subtitle's audio segment aligns accurately with the video's audio track. The determined offset is stored in `session.json` for subsequent processing and reference.

Aligning all subtitles

Upon successfully aligning the initial subtitle segment, the process extends to other subtitles within the video. The steps involved are:

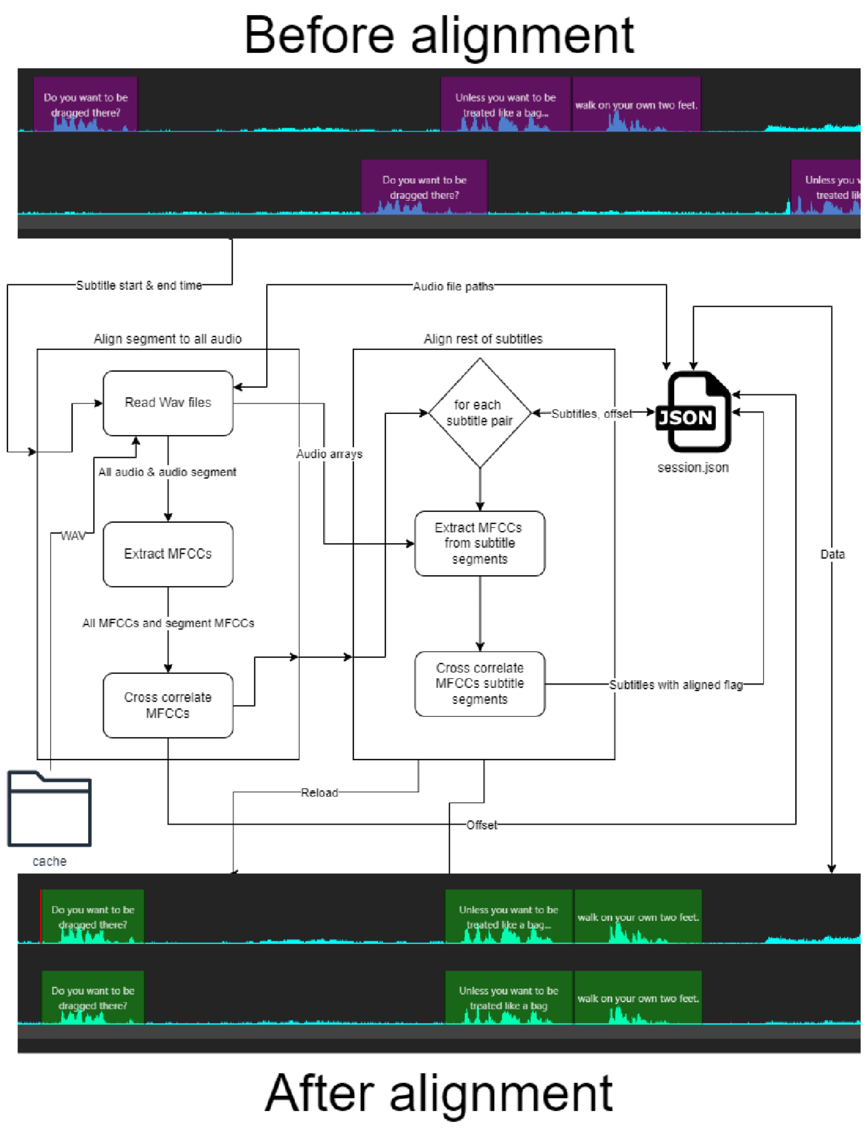

Before alignment
Do you want to be dragged there?
Unless you want to be treated like a bag...
walk on your own two feet.
Do you want to be dragged there?
Unless you want to be treated like
Subtitle start & end time
Audio file paths
Align segment to all audio
Align rest of subtitles
Read Wav files
for each subtitle pair
Subtitles, offset
JSON
session.json
Audio arrays
All audio & audio segment
WAV
Extract MFCCs
Extract MFCCs from subtitle segments
Data
All MFCCs and segment MFCCs
Cross correlate MFCCs subtitle segments
Subtitles with aligned flag
Cross correlate MFCCs
cache
Reload
Offset
Do you want to be dragged there?
Unless you want to be treated like a bag...
walk on your own two feet.
Do you want to be dragged there?
Unless you want to be treated like a bag
walk on your own two feet.
After alignment

Figure 4.7: Alignment process in the backend

- Evaluation of audio signals from other subtitles to ascertain if they match using a similarity threshold established through preliminary experiments.
- Identification of subtitle pairs that are temporally proximate, enhancing the likelihood of alignment.
- Extraction and cross-correlation of MFCCs from these subtitle segments. Successful alignment is indicated by an offset equal to zero and maximum correlation within the experimentally determined range.
- Subtitles that are aligned are flagged and adjusted in timing to ensure precise synchronization.
- The updated subtitles are saved into `session.json`.

- A notification is sent to the frontend, prompting it to refresh and display the aligned subtitles.

4.3 Production vs development

This section presents a comparative analysis between the operational environments of backend and frontend technologies in **Production** and **Development** contexts.

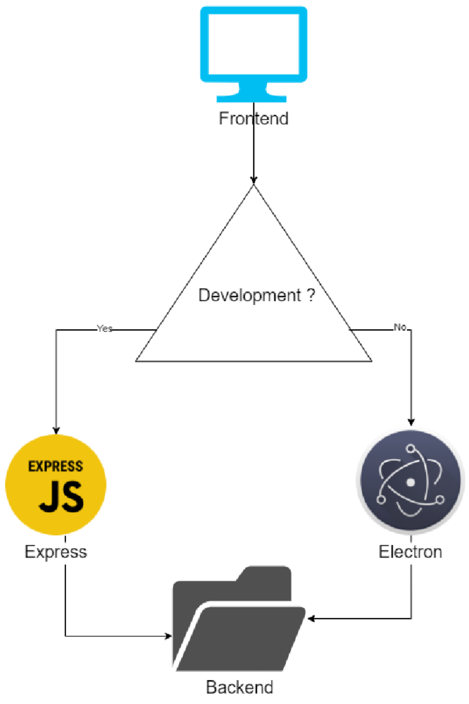


Figure 4.8: Production vs development frameworks

The overview of the frameworks used for production and development is showcased in figure 4.8. In the production setting, the system utilizes a built `Vue3` application combined with `Electron`. This setup ensures a stable and efficient application deployment suitable for end-users. During development, the system operates using a preview `Vue3` application that features hot module reload, facilitating immediate updates and debugging. An `Express` server is used as the backend, supporting dynamic development needs with increased flexibility.

The file `client.js` plays a crucial role in determining the interaction between the frontend and the backend services, either through the `Express` server or `Electron`'s IPC (Inter-Process Communication). Below is the code snippet showcasing how the application determines the environment and communicates accordingly:

```
import axios from 'axios'

const isElectron = typeof window.electron !== 'undefined'
const apiService = {
  async sendMessage(channel, data, config) {
    if (!isElectron) {
      if (config?.method === 'GET') {
        return await axios.get(`http://localhost:3000/${channel}`)
      }
      return await axios.post(`http://localhost:3000/${channel}`, data, config)
    }
    console.log('API request:', channel, data)
    const response = {
      data: await window.electron.ipcRenderer.invoke(channel, data),
    }
    console.log('API response:', response)
    return response
  },
}

export default apiService)
```

Listing 4.3.1: Endpoint for frontend - `client.js`

The `client.js` script shown in listing 4.3.1 checks if the application is running in an Electron environment (indicative of production) by verifying the presence of `window.electron`. Depending on the environment, it chooses between making HTTP requests via `axios` (for development) or using Electron's IPC features (for production) to communicate with the backend.

`Axios` is a promise-based HTTP client for making requests to external servers. In the development environment, `axios` is used to perform GET or POST requests to an `Express` server running locally. This is essential for rapid development and testing, allowing for real-time feedback and updates via hot module reloading.

In an Electron application, the `ipcRenderer` module as mentioned in 3 of Electron facilitates communication between renderer processes (web pages) and the main process. In this code, `electron.ipcRenderer` is accessed via the `window` object. This integration is achieved by modifying the `window` object in the Electron environment to include `ipcRenderer`, thus making it available as `window.electron.ipcRenderer`. This allows the frontend to send messages and receive responses from the main process, enabling seamless backend-frontend integration in a desktop application context.

Chapter 5

Testing

This chapter delves into the various testing methodologies employed to ensure the reliability, performance, and user experience of the application across different platforms and operational scenarios. It outlines the structure and purpose of compatibility and performance tests.

5.1 Test Data

This section introduces the various datasets utilized for testing the application, categorized into small and large data sets to accommodate different testing scenarios such as compatibility and performance testing.

Small datasets are crucial for compatibility testing to ensure that the application functions correctly across multiple environments. These datasets include:

- `example1.mkv`: A short video file containing speeches.
- `example1_extended.mkv`: An extended version of `example1.mkv`.
- `example1.srt`: Subtitles corresponding to `example1.mkv`.
- `example1_extended.srt`: Subtitles for the extended video `example1_extended.mkv`.

These files are designed to be lightweight and easily downloadable to various testing environments, with the download facilitated by a script provided in `README.md`.

To evaluate the application's performance, particularly the speed of alignment, larger datasets are used:

- `Maids.E01.e-subs.540p-ZASK.mkv`: A larger video file used to test the application's processing capabilities.
- `Maids.E01.e-subs.540p-ZASK.srt`: Subtitles for the aforementioned video.
- `Maids.S01E01.1080p.WEB-DL.H264.AAC-AppleTor.mp4`: An extended high-definition version of `Maids.E01.e-subs.540p-ZASK.mkv`.
- `Maids.S01E01.1080p.WEB-DL.H264.AAC-AppleTor.srt`: Subtitles for the extended HD video.

These files are significantly larger and contain private content, making them unsuitable for public download. They are used internally to assess how well the application handles large-scale data inputs and aligns long subtitle tracks efficiently.

5.2 Compatibility testing

Compatibility testing assesses whether the application functions as intended across various operating systems such as Linux, Windows, and Mac. This testing is facilitated by the automated testing framework `Playwright`, which allows for the simulation of user interactions with the application in different environments[1]. The utilization of `Playwright` is particularly effective due to the application's development in JavaScript, enabling full functionality within a web browser. This compatibility with web standards allows for a seamless integration of `Playwright`, enhancing the testing process across different platforms.

The primary goal of these tests is to ensure that the application behaves consistently and reliably regardless of the operating system or hardware configuration. This involves checking that all functionalities are accessible and perform as expected across all supported platforms.

```
"C:\Program Files\nodejs\npm.cmd" test

> my-electron-app@0.0.0 test
> playwright test --workers=1


Running 13 tests using 1 worker
  Slow test file: [Google Chrome] › example.spec.js (31.4s)
  Consider splitting slow test files to speed up parallel execution
  13 passed (32.6s)

To open last HTML report run:

  npx playwright show-report


Process finished with exit code 0
```

Listing 5.2.1: Running automated tests

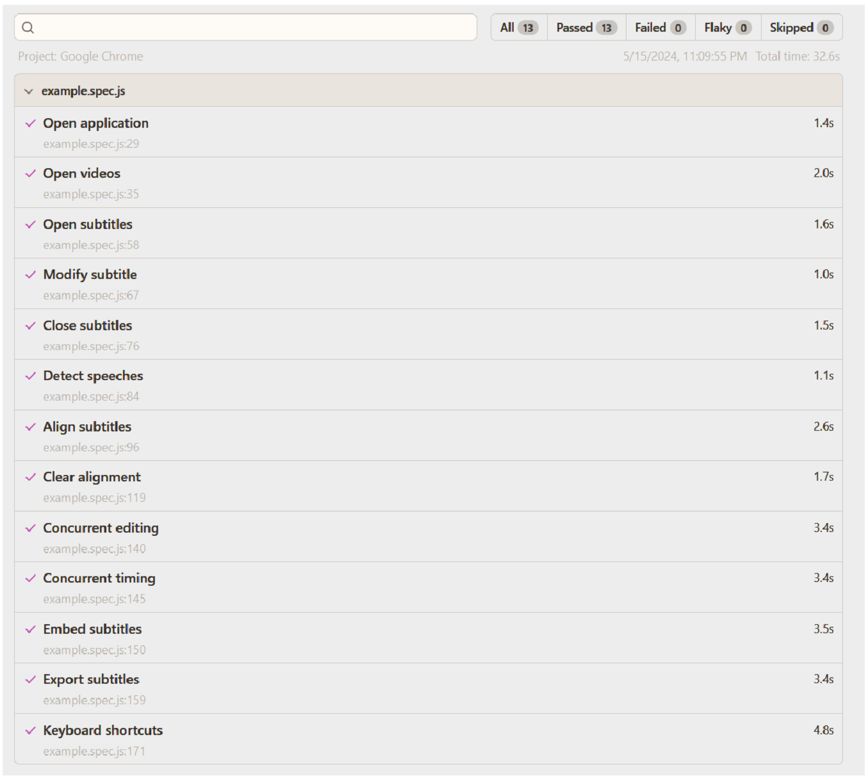
These tests were conducted on Linux, Windows, and Mac environments, successfully passing all the predefined test scenarios. The results from these tests are compiled into an HTML report, which is included in the `Playwright` test packages. An example of such a report is depicted in figure 5.1

The tests described utilize Playwright to automate interaction sequences in a development environment mimicking the functionality of the production build. Each test operates within a fresh instance of a Chrome browser, akin to opening and closing the application in `Electron`, with state persistence managed through `session.json`. This setup ensures that each feature can be tested in isolation, providing a reliable and consistent testing environment.

Here's a brief description of what each test accomplishes:

- **Open application** – Verifies that the application loads properly and displays the expected title, ensuring the initial launch is successful.
- **Open videos** – Tests the functionality of loading video files by interacting with the UI to select and verify the visibility of specific video files.

[1]For more details on Playwright, see the official documentation: `https://playwright.dev`



```
All 13   Passed 13   Failed 0   Flaky 0   Skipped 0
Project: Google Chrome                    5/15/2024, 11:09:55 PM  Total time: 32.6s

example.spec.js
✓ Open application                        1.4s
  example.spec.js:29
✓ Open videos                             2.0s
  example.spec.js:35
✓ Open subtitles                          1.6s
  example.spec.js:58
✓ Modify subtitle                         1.0s
  example.spec.js:67
✓ Close subtitles                         1.5s
  example.spec.js:76
✓ Detect speeches                         1.1s
  example.spec.js:84
✓ Align subtitles                         2.6s
  example.spec.js:96
✓ Clear alignment                         1.7s
  example.spec.js:119
✓ Concurrent editing                      3.4s
  example.spec.js:140
✓ Concurrent timing                       3.4s
  example.spec.js:145
✓ Embed subtitles                         3.5s
  example.spec.js:150
✓ Export subtitles                        3.4s
  example.spec.js:159
✓ Keyboard shortcuts                      4.8s
  example.spec.js:171
```

Figure 5.1: Automated tests - results

- **Open subtitles** – Ensures that subtitle files can be loaded and displayed correctly by selecting them through the UI and checking their presence on the screen.
- **Modify subtitle** – Confirms the ability to edit subtitles and verify that the changes are reflected within the application's interface.
- **Close subtitles** – Tests the capability to close subtitle files and ensures that they are removed from the display, verifying the close functionality.
- **Detect speeches** – Automates the detection of speech within video content and checks for the appropriate generation and display of subtitles.
- **Align subtitles** – Verifies that subtitles can be accurately aligned with the video's audio through UI interactions and evaluates the alignment by checking specific UI elements.
- **Clear alignment** – Tests the ability to remove existing alignments of subtitles, ensuring that the UI updates to reflect these changes correctly.
- **Concurrent editing** – Simulates the scenario of multiple users editing subtitles at the same time to test the application's handling of concurrent edits.
- **Concurrent timing** – Examines the application's capacity to manage simultaneous adjustments to subtitle timings by multiple users.
- **Embed subtitles** – Checks the embedding functionality by integrating subtitles into the video and ensuring no error messages appear during the process.

- **Export subtitles** – Validates the functionality to export subtitles successfully and checks for the presence and accuracy of exported files.
- **Keyboard shortcuts** – Assesses the responsiveness of the application to keyboard shortcuts for various subtitle editing tasks, enhancing user interface interaction.

5.3 Performance Testing

Performance testing involves custom manual testing by measuring the speed of subtitle alignment and the loading time of waveforms. These tests are essential for evaluating the efficiency and responsiveness of the application under different data loads and conditions. Test results in this section were created by measuring execution time of related functions.

5.3.1 Loading of Waveform

When a new video is opened, the application converts the audio track to a WAV file, a process that can be time-consuming, as shown in listing 5.3.1.

```
Audio extracted to: backend/cache/videos_Maids.E01.e-subs.540p-ZASK.wav
Converted \videos\Maids.E01.e-subs.540p-ZASK.mkv to Wav in: 12.6314983s
Audio extracted to: backend/cache/videos_Maids.S01E01.1080p.WEB-DL.H264.AAC-AppleTor.wav
Converted \videos\Maids.S01E01.1080p.WEB-DL.H264.AAC-AppleTor.mp4 to Wav in: 22.4528153s
```

Listing 5.3.1: Converting audio to WAV file

However, upon subsequent loads, the application utilizes the cached WAV files, significantly reducing loading times, as shown in listing 5.3.2.

```
Loaded cached WAV file: backend/cache/videos_Maids.E01.e-subs.540p-ZASK.wav
Converted \videos\Maids.E01.e-subs.540p-ZASK.mkv to Wav in: 0.0010645s
Loaded cached WAV file: backend/cache/videos_Maids.S01E01.1080p.WEB-DL.H264.AAC-AppleTor.wav
Converted \videos\Maids.S01E01.1080p.WEB-DL.H264.AAC-AppleTor.mp4 to Wav in: 0.0012184s
```

Listing 5.3.2: Loading of cached WAV files

The waveform for the video is then generated from the decoded audio using the `wavesurfer.js` library, as shown in listing 5.3.3.

```
Wavesurfer is ready. Loading took 2138 milliseconds.
Wavesurfer is ready. Loading took 2064 milliseconds.
```

Listing 5.3.3: Wavesurfer.js rendering speed

Although the overall performance is satisfactory, reliability is a concern. Approximately 5% of the time, the waveform is not visible due to issues with the `wavesurfer.js` library. While efforts have been made to address this bug, a complete fix has not been achieved. A simple workaround is to adjust the wave zoom to a lower value and then back to a higher value, which restores the waveform visibility. This bug is displayed in figure 5.2.

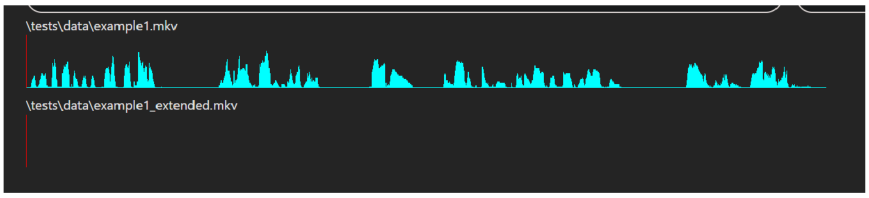


Figure 5.2: Waveform loading error

5.3.2 Alignment of Subtitles

The speed of subtitle alignment varies with the signal values of the reference subtitle, as shown in listings 5.3.4 and 5.3.5.

```
Segment length: 17592
Audio length: 33503915
MFCC segment length: 34
MFCC signal length: 65437
Best offset: 81.34400000000001
Signals aligned in: 11.509625400000001s
MaxCorr 69293.09164471625
All subtitles aligned in: 0.2602723s
```

Listing 5.3.4: Aligning a 2.2-second subtitle

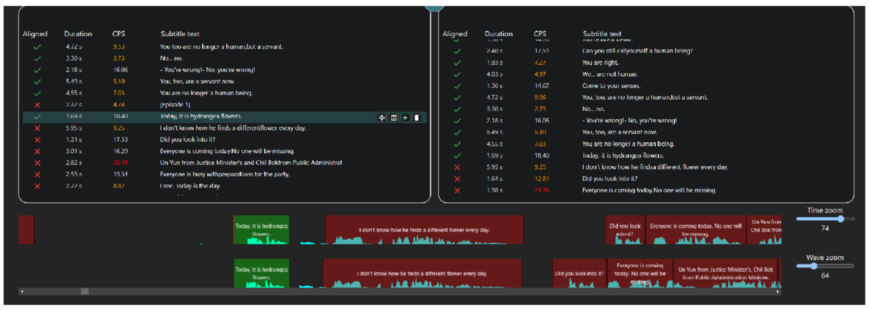
Command shown in listing 5.3.4 aligns a reference subtitle of 2.2 seconds.

```
Segment length: 25888
Audio length: 33503915
MFCC segment length: 50
MFCC signal length: 65437
Best offset: 176.576
Signals aligned in: 10.664324s
MaxCorr 154863.00279628698
All subtitles aligned in: 0.2302391s
```

Listing 5.3.5: Aligning a 3.2-second subtitle

Command shown in listing 5.3.5 aligns a reference subtitle of 3.2 seconds.

The alignment process is highly accurate when the subtitle segment is part of the audio, which it is aligning to. However, if it is not, the resulting best correlation can produce inaccurate alignments. Most subtitles are aligned correctly, but occasionally, the high correlation of signal pairs results in subtitles not being marked as aligned. This issue is illustrated in figure 5.3.



Aligned	Duration	CPS	Subtitle text
✓	4.72 s	9.53	You too are no longer a human,but a servant.
✓	3.30 s	2.73	No... no.
✓	2.18 s	16.06	- You're wrong!- No, you're wrong!
✓	5.49 s	5.10	You, too, are a servant now.
✓	4.55 s	7.03	You are no longer a human being.
✗	2.32 s	4.74	[Episode 1]
✓	1.69 s	10.40	Today, it is hydrangea flowers.
✗	5.95 s	9.25	I don't know how he finds a differentflower every day.
✗	1.21 s	17.33	Did you look into it?
✗	3.01 s	16.29	Everyone is coming today.No one will be missing.
✗	2.82 s	28.41	Un Yun from Justice Minister's and Chil Boktrom Public Administrat
✗	2.53 s	10.34	Everyone is busy withpreparations for the party.
✗	2.72 s	8.82	I see. Today is the day.

Aligned	Duration	CPS	Subtitle text
✓	2.40 s	17.51	Can you still callyourself a human being?
✓	1.93 s	7.27	You are right.
✓	4.03 s	4.97	We... are not human.
✓	1.36 s	14.67	Come to your senses.
✓	4.72 s	9.96	You, too, are no longer a human,but a servant.
✓	3.30 s	2.73	No... no.
✓	2.18 s	16.06	- You're wrong!- No, you're wrong!
✓	5.49 s	5.10	You, too, are a servant now.
✓	4.55 s	7.03	You are no longer a human being.
✓	1.69 s	10.40	Today, it is hydrangea flowers.
✗	5.95 s	9.25	I don't know how he findsa different flower every day.
✗	1.64 s	12.81	Did you look into it?
✗	1.98 s	24.76	Everyone is coming today.No one will be missing.

Today, it is hydrangea flowers.
I don't know how he finds a different flower every day.
Did you look into it?
Everyone is coming today. No one will be missing.
Un Yun from Justice Minister's, Chil Bok from Public Administration Minister.
Time zoom
74
Wave zoom
64

Figure 5.3: Alignment inaccuracy

In summary, while performance testing indicates good speed and efficiency in loading and alignment processes, certain reliability issues with waveform visibility and subtitle alignment accuracy need to be addressed to enhance the overall performance and user experience.

Chapter 6

Conclusion

This thesis has outlined the design, development, and testing of a multi-platform tool aimed at simplifying the process of editing subtitles for videos that have different versions. By addressing the unique challenge of synchronizing subtitles across various cuts and inserted scenes, this tool offers a significant advancement in the realm of subtitle editing.

6.1 Summary of Achievements

The primary objective of this work was to create a tool that allows for the simultaneous editing of multiple subtitle files, which is particularly useful for videos with different versions. This objective has been successfully met through several key achievements:

- **Cross-Platform compatibility**: By leveraging frameworks such as Electron (2.1.4) and Tauri (2.1.4), the tool ensures a consistent user experience across major operating systems, including Linux, Windows, and Mac.
- **Subtitle and video format support**: The tool supports multiple subtitle formats, including SRT and WebVTT as discussed in (2.2). Supported video formats are MP4 and MKV. This flexibility ensures broad applicability for various user needs.
- **Advanced signal processing**: Incorporating audio-visual signal processing techniques such as waveform analysis (2.3.2), Dynamic Time Warping (DTW), Mel-frequency cepstral coefficients (MFCC), and Fast Fourier Transform (FFT) allows for accurate synchronization of subtitles. The usage of FFmpeg for transcoding (2.4.4) and subtitle extraction (2.4.4) further enhances the tool's capabilities.
- **User-Friendly interface**: The design and implementation of a user-friendly interface, as detailed in (3.1) and (3.5), ensure that both novice and experienced users can efficiently use the tool for subtitle editing.
- **Robust testing framework**: Extensive compatibility and performance testing, facilitated by tools like Playwright, have validated the tool's reliability across different environments. The detailed testing results, including performance metrics for subtitle alignment and waveform loading, demonstrate the tool's efficiency and effectiveness.

6.2 Challenges and Limitations

While the tool has achieved its primary goals, several challenges and limitations were encountered:

- **Waveform Loading Reliability**: Despite overall satisfactory performance, the waveform loading process occasionally fails due to issues with the `wavesurfer.js` library (5.2). Although a workaround exists, further refinement is needed for a permanent fix.
- **Alignment Accuracy**: While the subtitle alignment process is generally accurate, there are instances where high correlation of signal pairs leads to misalignment (5.3). Continuous improvement of the alignment algorithms is necessary to enhance reliability.

6.3 Future Work

Looking ahead, several areas for future development and improvement have been identified:

- **Enhanced Bug Fixing**: Addressing the reliability issues with waveform visibility and alignment accuracy will be a priority. This involves further development and testing of the `wavesurfer.js` library and the alignment algorithms.
- **Community Feedback and A/B Testing**: As the tool is released as an open-source project, gathering feedback from the community will be invaluable. Implementing A/B testing will help refine the user interface and functionality based on real-world usage data.
- **Expanded Feature Set**: Adding new features such as advanced spell checking, more robust speech recognition capabilities, and additional subtitle formats will further increase the tool's utility and appeal.
- **Performance Optimization**: Continued efforts to optimize the tool's performance, particularly for large datasets, will ensure that it remains responsive and efficient even under heavy usage conditions.

6.4 Summary

In conclusion, this thesis has successfully demonstrated the development of a comprehensive, user-friendly tool for simultaneous subtitle editing across different video versions. By addressing the specific challenges of subtitle synchronization and leveraging advanced signal processing techniques, the tool provides a powerful solution for video editors and content creators. The ongoing development and community engagement will further enhance its capabilities, ensuring it remains a valuable resource in the field of multimedia editing.

Bibliography

[1] ADOBE. Choosing the Best Audio Format. [online]. 2024. Accessed: April 30, 2024. Available at: https://www.adobe.com/creativecloud/video/discover/best-audio-format.html.

[2] ADOBE. Choosing the Best Video Format. [online]. 2024. Accessed: April 30, 2024. Available at: https://www.adobe.com/creativecloud/video/discover/best-video-format.html.

[3] C. A. BULTERMAN, D., JANSEN, J., CESAR, P. and CRUZ LARA, S. An Efficient, Streamable Text Format for Multimedia Captions and Subtitles. In: *ACM Symposium on Document Engineering - DocEng 2007*. [b.n.], August 2007. Available at: https://inria.hal.science/inria-00192467.

[4] CONTRIBUTORS, W. *Audio file format* [online]. Wikipedia, The Free Encyclopedia, 2024. Date of last revision: 15 May 2024 09:02 UTC. Accessed: April 30, 2024. Date retrieved: 16 May 2024 06:07 UTC. Available at: https://en.wikipedia.org/w/index.php?title=Audio_file_format&oldid=1223942973.

[5] CONTRIBUTORS, W. *Video file format* [online]. Wikipedia, The Free Encyclopedia, 2024. Date of last revision: 10 May 2024 19:28 UTC. Accessed: May 16, 2024. Date retrieved: 16 May 2024 06:08 UTC. Available at: https://en.wikipedia.org/w/index.php?title=Video_file_format&oldid=1223237686.

[6] KAPURIYA, P. Desktop Application Development: A Complete Guide. Codz Garage. 2023. [Last edited: 4/15/2024]. Available at: https://www.codzgarage.com/blog/desktop-app-development-guide/.

[7] MÜLLER, M. *Information Retrieval for Music and Motion*. 1st ed. Springer, 2007. 69–84 p. ISBN 978-3-540-74048-3.

[8] NIKKHAH, E. Subtitles file formats: The most used and when to choose each one. [online]. 2023. Accessed: May 3, 2023. Available at: https://scriptme.io/subtitle-file-formats/.

[9] NUNES, E. C. Anomalous Sound Detection with Machine Learning: A Systematic Review. *Ar5iv*. 2021. Accessed: May 1, 2024. Available at: https://ar5iv.labs.arxiv.org/html/2102.07820.

[10] OPPENHEIM, A. V. and SCHAFER, R. W. *Discrete-time Signal Processing*. 2nd ed. Upper Saddle River, NJ: Pearson, dec 1998. ISBN 0-13-754920-2.

[11] Oppenheim, A. V. and Willsky, A. S. Continuous-time and Discrete-time Signals. In: Horton, M., ed. *Signals and Systems*. 2nd ed. Upper Saddle River, NJ: Pearson, Aug 1996, chap. 1.1. ISBN 0-13-814757-4.

[12] Ozer, J. *Video Codecs in 2021 and Beyond* [Streaming Learning Center Webinar]. 2021. Accessed: May 1, 2024. Available at: https://streaminglearningcenter.com/learning/webinar-video-codecs-in-2021-and-beyond-with-jan-ozer.html.

[13] Company, Q. Introduction to Qt. Qt. 2024. [Accessed on: 1/5/2024]. Available at: https://doc.qt.io/qt-6/qt-intro.html.

[14] Singh, P. P. and R., P. An Approach to Extract Feature using MFCC. *IOSR Journal of Engineering*. IOSR Journals. august 2014, vol. 4, no. 8, p. 21–25. DOI: 10.9790/3021-04812125. Available at: https://doi.org/10.9790/3021-04812125.

[15] TechTerms. *Waveform Definition* [online]. 2015. Retrieved: December 9, 2015. Available at: https://techterms.com/definition/waveform.

[16] Cloud, V. What is a Desktop App? V2 Cloud. 2024. [Accessed on: 1/4/2024]. Available at: https://v2cloud.com/glossary/what-is-a-desktop-app/.

[17] Roca, J. What is Cross Platform Software? Triangle. 2022. [Accessed on: 1/5/2024]. Available at: https://www.triangle.es/en/what-is-cross-platform-software/.