Bauhaus-Universität Weimar
Faculty of Media
Degree Programme Medieninformatik

LSML: Generating Spoken Lectures From Scripts

Bachelor's Thesis

Christian Dunkel
Born June 9, 1997 in Saalfeld

Matriculation Number 117202

1. Referee: Prof. Dr. Benno Stein
2. Referee: Prof. Dr. Jan Ehlers

Submission date: October 19, 2020

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Weimar, October 19, 2020

. . . .
Christian Dunkel

Abstract

In this thesis, I outline my process of planning and developing the tool lecture.js that allows teachers to generate spoken lectures using only a script and slides. Current Text-to-Video software focuses mainly on providing massive databases of video clips and images which are automatically selected and matched to the text. I concentrate on writing a tool designed explicitly for Script-to-Lecture generation. For that purpose, I develop the Lecture Synthesis Markup Language (LSML) as an extension of the Speech Synthesis Markup Language (SSML). LSML enables teachers to write lecture scripts, define the voice output and video settings, and control their slides in the same document. The tool then processes the document and communicates with multiple advanced Text-to-Speech services to produce natural-sounding speech using machine learning. In the end, the received audio files will be stitched together with the slides to generate a video lecture.

Contents

List of Figures

Acknowledgements

I thank Jun.-Prof. Martin Potthast and Lars Meyer for guiding me through writing this thesis and dedicating their time and effort in helping me plan and improve the software project. Without them, this thesis would not have been possible.

I also want to give Christian Hilpert special gratitude, who sat with me through lengthy bug-finding sessions.

Glossary

API An Application Programming Interface is an interface that allows access to the services of another software that implements that API.

HTML Hypertext Markup Language is an XML-based markup language for documents displayed in web browsers.

IPA The International Phonetic Alphabet is the most widespread phonetic alphabet used in phonetic transcription. It is used to precisely define the pronunciation of a word.

JSON JavaScript Object Notation is a standardized, human-readable file format for saving and transmitting data across a number of different languages, services, programs, and operating systems.

LSML Lecture Synthesis Markup Language is an extension of Speech Synthesis Markup Language (SSML) developed in this thesis. It is intended to support aspects of Text-to-Speech and Text-to-Speech.

PDF Portable Document Format is a file format that focuses on making shareable documents, which include images and formatted text, software- and hardware-independent.

SSML Speech Synthesis Markup Language is an XML-based markup language for speech synthesis applications that generalizes the process of speech synthesis by providing a language specification suitable for different platforms.

UML The Unified Modeling Language is a modeling language intended to standardize system designs.

URI A Uniform Resource Identifier is a string that identifies a particular resource unambiguously.

URL A Uniform Resource Locator is a string that references the location of a web resource on a computer network.

UUID A Universally Unique Identifier is a 128-bit number used to uniquely identify data.

W3C The World Wide Web Consortium is an organization that sets the international standards for the World Wide Web.

XML Extensible Markup Language is a markup language that defines rules for encoding documents in a format that is human- and machine-readable.

XSD XML Schema Definition is a type of schema definition that describes the elements in a XML document, which is then often used for validation.

Chapter 1

Introduction

In today's digital world, it is becoming increasingly important to integrate digitalization into the classroom to reap the advantages of greater interactivity and distance learning. This importance only grew in light of the recent COVID-19 pandemic, which necessitated social distancing and quarantine worldwide. Globally, 1.2 billion learners were affected by 150 countries imposing country-wide closures in May 2020, which made up 68% of total enrolled learners, as reported by UNESCO [2020].

The COVID-19 pandemic forced teachers to embrace e-learning technology to continue conducting classes. Consequently, software for e-learning and web communications experienced substantial growth. Zoom, which is a software for video conferencing, recorded daily downloads of 56 thousand on February 23rd, 2020. Two months later, on March 23rd, they had an increase of 3800% to 2.13 million downloads, when the UK's lockdown was announced, as reported by The Guardian [2020]. The conference software Big Blue Button also experienced a boom at German schools and universities. For example, in June 2020, an estimated 500 schools in Baden Württemberg alone employed the software, as stated by the Ministry for Culture, Youth and Sport of Baden Württemberg [2020]. If those numbers are any indication, the rise of popularity for e-learning software in the first two quarters of 2020 may only be the beginning. It is essential to introduce tools that enable learners and teachers worldwide to create and consume lectures efficiently.

Live streaming the classroom is steadily gaining popularity, but another type of e-learning has also been a staple of education for over a decade. Prerecorded lectures present unique benefits for learners. First and foremost, students appreciate them, found by Couperthwaite et al. [2012], Nordmann and McGeorge [2018]. Students perform better at memory exam questions when using them, but the comprehension of the material may suffer if they do not access the lectures on time before an exam, found Hadgu et al. [2016].

Students can adjust the playback speed on the pre-recorded lectures, replay sections, pause and look up additional information on the side to better understand a difficult subject. Recorded lectures also present the advantage of closed captioning for students with hearing disabilities or students who listen to the lecture in a language that is not their native tongue.

However, the adoption of pre-recorded lectures also comes with a new set of problems. The teacher has to invest a lot more time into additional tasks unrelated to the lecture's content. Teachers have to film their slides, record their script, replace sub-par voice segments, and edit the video until they can render and upload it to the internet. Audio and video editing require costly software, hardware, and know-how to produce quality content. But why go through all that trouble when a computer could automate most of the menial tasks of video production? Teachers would have more time to focus on writing the script and creating appealing slides if a machine could produce the video.

The need to easily generate lectures comes at a time when personal voice assistants are becoming more sophisticated, and voices generated using artificial intelligence sound more and more life-like. Since 2016, Amazon is working on its cloud service Amazon Polly that converts text into life-like speech.[1] Google also offers its services for generating speech from text using its Google Cloud platform with hundreds of voices in dozens of different languages.[2]

Each of these speech synthesis services offers an Application Programming Interface (API) that can be integrated into a new software project and provide a way to convert text to speech programmatically. Most importantly, these services support SSML, a markup language for speech synthesis that allows users to control the pitch, speed, and even pronunciation of words. Since SSML is a language based on the Extensible Markup Language (XML), it can be effortlessly extended to handle the visual aspect of the final video lecture, like changing the slide at a particular section. These features allow for developing a tool that enables teachers to generate complete video lectures from just a script and slides.

In this thesis, I will discuss my process of extending the SSML language by creating the Lecture Synthesis Markup Language (LSML) for lecture-generation and implementing the Script-to-Lecture software lecture.js. In the end, I will conduct a user study to evaluate the software and analyze the results.

[1] Amazon Polly: aws.amazon.com/polly/

[2] Google Cloud Text-to-Speech: cloud.google.com/text-to-speech

Chapter 2

Related Work

Students generally hold positive attitudes towards pre-recorded lectures and tend to use them in a targetted manner, as found by Couperthwaite et al. [2012]. In the study, 50-75% of the student body accessed the pre-recorded lectures, often selecting lectures with the most complex topics and taking notes. Dyslexic students particularly utilized the material.

However, Danielson et al. [2014] discovered a negative relationship between the interactivity of a lecture and the likelihood that students would watch the pre-recorded lectures. Students tend to watch a higher number of lectures for courses that rely more heavily on the lecture and less on interactivity. The study attributes this to the fact that pre-recorded lectures primarily capture the teacher's and not the student's actions like student questions, which give interactive lectures add a lot of their value. These findings suggest that pre-recorded lectures for interactivity-heavy courses can not replace live lectures.

In the same study, over 90% of the students self-assessed that they were likely to learn better using pre-recorded lectures. Hadgu et al. [2016] also found that student performance, when learning with pre-recorded lectures compared to live lectures, improved for answering memory questions, which require the memorization of basic factual details. However, students seemed to fare better with comprehension questions when they attended live lectures, which required understanding the taught information.

This research suggests that pre-recorded lectures are not a substitute for live lectures. However, they can be a useful supplement for the teaching process. The effectiveness of hybrid course models consisting of both live lectures and pre-recorded lectures, is also supported by Prunuske et al. [2012]. They carried out a study that tested student performance when providing pre-recorded lectures in addition to homework assignments. These pre-recorded lectures helped students self-reportedly to complete their assignments without increasing the time expenditure for the course.

The study ran for five years, in which 70-85% of the participating students watched the pre-recorded lectures before attending the live lectures in class, and 97-99% of students planned to do so before the examination.

The volume of learning material, especially in university lectures, can often not be wholly comprehended in the live lectures alone. Taking complete notes may also be difficult if one tries to understand advanced concepts at the same time. The research suggests that this is where pre-recorded lectures can help students comprehend learning material and noticeably improve learning results. Still, creating video lectures besides live lectures can be time-consuming. Although teachers may significantly reduce the time cost if there exists software to automate parts of the process.

A multitude of commercial and open-source Text-to-Video software is available on the market. However, these focus mainly on providing massive databases of video clips and images, which are automatically selected and matched to the user's text. For example, RawShorts[1] and Lumen5[2] are services that take raw text and convert it to speech, then automatically match it to video clips and images from a database to generate animated videos. Besides not being commercially free, both services also do not allow for the precise control required for generating a lecture. These services are more suited to prototype a video idea.

There also exist solutions for converting articles to videos. The services Article Video Robot[3] and Viomatic[4] convert online articles to videos with speech. A user can enter the Uniform Resource Locator (URL) of an article, and the tool will grab the text and images from the web page. However, both services are not commercially free, and the resulting videos are not very user-customizable. Besides, creating a video lecture this way is very roundabout and inefficient. It could be challenging to match the article's images, which would be the corresponding lecture slides, precisely to the right text sections. GliaStudio[5] works similarly but automatically creates video summaries of online articles. After entering the URL to a web page or uploading a text document, a natural language algorithm will try to find important topics and keywords in the text. The algorithm will then generate video scripts with sections and highlights. Based on these, an AI will search for images and video clips and edit them together. Needless to say, GliaStudio is not suited for generating lectures since leaving out any part of the lecture would hide information necessary for understanding the lecture's contents.

[1] RawShorts: rawshorts.com
[2] Lumen5: lumen5.com
[3] Article Video Robot: articlevideorobot.com
[4] Viomatic: viomatic.com
[5] GliaStudio: gliacloud.com

Among the surveyed solutions, the one most suitable for generating lectures seems to be the commercial service Narakeet[6], formerly known as Videopuppet. Narakeet is an online service that automatically transforms scripts into speech and synchronizes them with images provided by the user. Narakeet also supports SSML, but only as a secondary part of its custom simplified non-XML markup language. Narakeet also only supports slides from presentation programs like Microsoft PowerPoint, Google Slides, or Apple Keynote. It either reads out the contents of a slide or the contents of the speaker notes attached to the slide as speech. It provides no support for Portable Document Format (PDF) documents, which are often utilized in university lectures as slides.

In contrast to the aforementioned available Text-to-Video implementations, existing Text-to-Speech software can be integrated into lecture.js and lend a solid foundation to the aspect of speech generation of Script-to-Lecture software. There are commercial Text-to-Speech solutions provided by Amazon Web Services and Google, called Amazon Polly and Google Cloud Text-to-Speech. Together they provide over 300 narrating voices in almost 40 different languages. Each of them also offers extensive APIs for many programming languages and environments, making them easy to integrate into the new software. There also exists the open-source framework OpenMARY[7], a multilingual speech synthesis system, originally developed by the German Research Centre for Artificial Intelligence in collaboration with the Institute of Phonetics at Saarland University. The framework supports the languages German, English, French, and Italian, amongst others. However, the voices provided by OpenMARY are not nearly as natural-sounding as the commercial alternatives mentioned above, which is why the OpenMARY project was not yet integrated into lecture.js.

[6] Narakeet: narakeet.com
[7] OpenMARY: mary.dfki.de

Chapter 3

Requirements

The Script-to-Lecture software's primary set of requirements includes features that make the software useful in day-to-day work for the most common user groups. The leading target user group for the software are teachers, lecturers, and other presenters. In this thesis, this user group is referred to as teachers. The primary requirements listed below were all met in the implementation of the software.

1. **Time-saving**: Teachers with little video editing experience need time to learn video and audio technology to record and edit videos. If a teacher can reduce or eliminate those tasks, which are not directly related to the lecture's content, they can spend more time working on the lecture itself.
2. **Cost-saving**: Teachers can use the software without acquiring expensive equipment, which includes hardware like a microphone and video editing software.
3. **Integration for external resources**: Teachers can embed external media resources besides slides into the lecture, like image, audio, and video files. External resources help the lecture remain dynamic, for example, by featuring screen recordings. A teacher must have control over which page of which slide or which other resource is visible at any time.
4. **Voice and language control**: Suppose teachers want to tell stories with different viewpoints in their lectures, they can switch the voice narrating the lecture at any point to recreate dialogue realistically or to highlight quotes. They can also control the language the narrating voice is using at any point in the script. Teachers can include words, segments, or whole sentences in another language, which is especially helpful since, in most languages, English words are adopted more frequently with increasing globalization.

5. **Manipulable pronunciation**: The language processor should automatically make good choices when generating speech. If it fails, however, teachers can correct the pronunciation of words. For example, the language processor may automatically detect the wrong language for a word like "information", which exists in both German and English. Teachers can then manually customize how the narrating voice pronounces "information" for specific sections or the whole of the script.
6. **Efficient editing**: Teachers may regularly rewrite their scripts. They can quickly change parts of the script or the slides and generate a new video without much effort and without amassing huge costs from the Text-to-Speech services.
7. **Self-containedness**: The software is stable and as self-contained as possible so that teachers can use it for years with as little maintenance and dependence on external tools as possible.
8. **Extendability**: Developers working on the software can rewrite significant portions of the source code to add new features or change behaviors. For this purpose, extensive documentation of the features and a modular program structure are required.

There is also a set of supplemental features listed below that are not essential for the software's core functionality but improve the user experience. However, only the first of the following features could be realized in time when writing this thesis. The other features should be noted down for future work on the software.

1. **Video Upload**: YouTube is an excellent platform for video hosting, as it is widely utilized, being the world's second most visited website, as measured by Alexa Siterank Competitive Analysis [2020], and the world's most popular video platform.[1] YouTube's Data API allows teachers to automatically upload generated video lectures to YouTube and assign them to playlists.
2. **Highlighting**: Sections of the slides can be highlighted to improve interactivity without teachers having to do it manually in the PDF document.
3. **GUI**: Teachers with no programming experience can use the software with a graphical interface without having to learn LSML.
4. **Question-Answering**: Students have access to an interface that augments the generated lectures with question-answering capabilities related to the lecture's contents.

[1] YouTube is also extensively utilized by the Webis group for publishing recorded lectures and talks on their channel: youtube.com/channel/UCGOTkqgnKS5a3bzzp1zU2Uw

5. **Subtitles**: For deaf or hard-of-hearing students, the software automatically adds subtitles to the generated video lecture. It offers the choice of burnt-in subtitles or closed captions.

In consideration of all primary requirements and supplemental features, figure 3.1 shows a use case diagram created using the Unified Modeling Language (UML). The diagram models the two main actors, teachers and students, and their primary use cases for the software. An extending use case (`«extends»`) depends on the base use case to which it points, while base use cases incorporate the behavior of the included use cases to which they point (`«include»`).

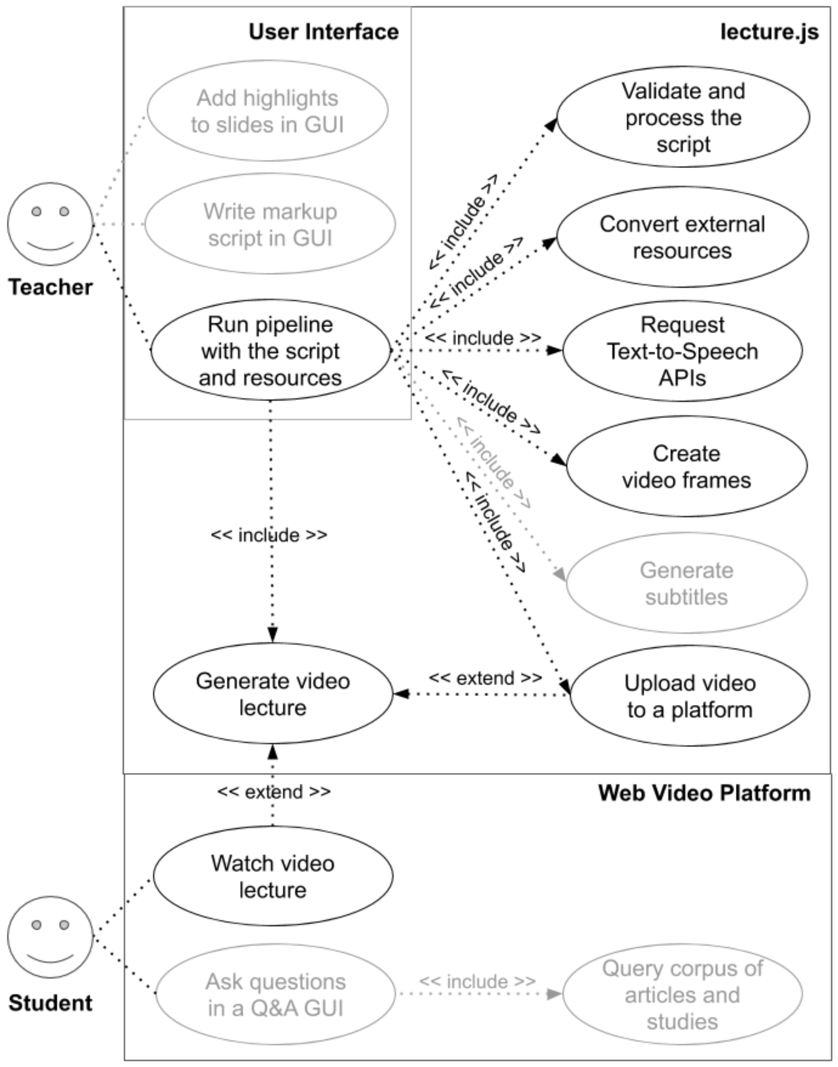

User Interface
lecture.js
Add highlights to slides in GUI
Write markup script in GUI
Teacher
Run pipeline with the script and resources
Validate and process the script
Convert external resources
Request Text-to-Speech APIs
Create video frames
Generate subtitles
Upload video to a platform
Generate video lecture
<< include >>
<< extend >>
Web Video Platform
Watch video lecture
Ask questions in a Q&A GUI
Query corpus of articles and studies
Student

Figure 3.1: UML Use Case Diagram

Chapter 4

LSML Language Definition

Lecture.js can not use a simple text document as the input script because the user needs to have sufficient control over various aspects of the lecture-generation process. For that purpose, this chapter proposes a custom markup language specification called Lecture Synthesis Markup Language (LSML).

LSML will use the syntax of XML as a fundament. XML is a hierarchically structured markup language that is both human- and machine-readable, which makes it perfect for an environment where both machines and possibly non-tech-savvy people work on the same document. XML offers many advantages; for example, there are multiple libraries available that support the parsing and transformation of XML markup. There also exist several technologies that extend XML, for example, XML Schema Definition (XSD) for validating XML documents. XSD will be very useful in specifying the structure of input documents and validating them in the software implementation. However, the most significant advantage of XML, in terms of speech synthesis, is that an XML-based language used for speech synthesis already exists. This language is the Speech Synthesis Markup Language (SSML) specified by the World Wide Web Consortium [2010], which can serve as a fundament of the custom language LSML to be extended with additional features.

SSML generalizes the process of speech synthesis by providing a language specification suitable for different environments. The language is based on indications rather than absolutes, meaning that the interpretation of markup values depends on the speech synthesizer that renders the content. This focus on generality makes SSML great for a software that requires the integration of multiple Text-to-Speech APIs. It is supported by many speech synthesis applications, which include Amazon Polly and Google Cloud Text-to-Speech, both of which will be integrated into lecture.js. However, there exist several differences between the SSML specification and the SSML implementations of those two APIs.

Additionally, SSML can only control the speech aspect of lecture generation, which is insufficient for lecture.js. LSML will accommodate for those limitations, and extend SSML by additional elements for controlling the visual aspect of Script-to-Lecture generation. In most aspects however, LSML will stay close to the specification of SSML v1.1. This has the advantage of making LSML more consistent in usage across different Text-to-Speech implementations since the SSML standard rarely changes.

In the following sections, I will summarize the core features of SSML as specified in the SSML Specification v1.1 by the World Wide Web Consortium [2010]. I will detail how Amazon Polly and Google Cloud Text-to-Speech, as well as the custom language LSML in relation to them, implement those features. I will also explain how LSML extends the language with additional Text-to-Video controls. Additionally, a detailed list of all elements and attributes with the level of support in different APIs can be found in the appendix starting on page XII. A simple LSML example script is also attached on page IV.

4.1 Document Structure

SSML Root Element

Every XML-based document is structured like a tree, with one root element containing all other elements. SSML being XML-based begins with an XML Prolog defining the XML version, followed by the `speak` element, which is the root element. The `speak` element defines specific information about the document. For example, the attribute `version` sets the SSML version that was used to write the document, and the attribute `xml:lang` sets the default language of the document. Additionally, there are also options to define a schema and namespace.

```
<?xml version="1.0"?>
<speak version="1.1" xml:lang="en-US">
    <!-- content -->
</speak>
```

The `speak` element can also determine where to start and stop rendering the SSML content by designating markers within the content as a start or end mark. Markers can be set using the `mark` element, and speech synthesizers can also use them to give users the option to retrieve timestamps at specific points in the document. Any content that comes before the marker designated as the `startmark` or after the marker set as the `endmark` will not be rendered.

```
<speak version="1.1" startmark="mark1" endmark="mark2">
    This sentence will be ignored.
    <mark name="mark1"/>
    This sentence will be spoken.
    <mark name="mark2"/>
    This sentence will be ignored.
</speak>
```

LSML Root Element

In LSML, however, the root element is different from the `speak` element in SSML. The lecture.js pipeline will need to request the APIs with SSML data as a string. Because of this and the XML validation using a custom XSD schema in the lecture.js pipeline, the `speak` element will not have a use for attributes defining schemas and namespaces. Additionally, the two Text-to-Speech APIs both implement a `speak` element; however, they only use it as a marker to identify SSML content and do not implement any of its attributes.

To compensate for these limitations, LSML does not support the `speak` element, but specifies an equivalent element with the different name `lecture` to differentiate the LSML markup from SSML. Because lecture.js will preprocess the markup content before sending it to the APIs, LSML also specifies the attributes `startmark` and `endmark`, which are not supported by the APIs.

LSML consequently fully supports the `mark` element for setting markers. Additionally, it specifies a `chapter` attribute, which creates a new chapter with the provided value as the name at the given marker. The pipeline can later use this information for defining a table of contents with timestamps.

```
<lecture startmark="intro">
    This sentence will be ignored.
    <mark name="intro" chapter="Introduction" />
    This is the introduction.
    <mark name="rel-work" chapter="Related Work" />
    This chapter is about the related work.
<lecture>
```

Meta Data

SSML specifies the elements `meta` and `metadata` for defining meta-information about the document. However, both elements are not supported by the APIs, and consequently, LSML does not support them either. To compensate for these missing elements, LSML specifies a custom element for holding meta information, which is the `info` element.

It is an empty element that may only appear as a direct child of the `lecture` element and may at most appear once. However, it and all of its attributes are optional. The element may define the `title` of the lecture, a `description`, a semicolon-separated list of `authors`, and additional `copyright` information.

```
<info
    title="An example lecture name"
    description="A short description of the lecture"
    authors="Max Mustermann; Erika Musterfrau"
    copyright="2020 Max Mustermann, Example University"
/>
```

Settings

LSML implements the element `settings` to manage all settings for an individual lecture. Like the `info` element, `settings` may only appear as a direct child of the `lecture` element and may at most appear once. All settings are optional and defined using an attribute-value pair.

```
<settings
    voice="amazon-de-de-vicki"
    resolution="1280x720"
    fps="30"
/>
```

If a setting is not defined, a default value from the configuration file is used instead. The following settings are supported:

- `voice` specifies the ID of the default voice for the document. The voice is used in sections where no other voice is specified.
- `resolution` defines the resolution of the video lecture in the format "*{width}x{height}*", for example, "*1280x720*" for High Definition.
- `fps` defines the number of frames per second for the resulting video as an integer.
- `breakAfterSlide` defines a break in milliseconds that should be applied by default when the slide changes.
- `breakAfterParagraph` defines a break in milliseconds that should be applied by default between all paragraphs.

- `googleEffectProfile` defines an effect profile[1] for all Google Cloud Text-to-Speech voices that are used in the document. Effect profiles are a feature of Google Cloud to optimize the generated speech for the playback on different types of hardware.
- `youtubePrivacyStatus` defines the privacy status of the video lecture, should the user upload it to YouTube using the tool. It supports the modes `public`, `unlisted` and `private`.
- `youtubePlaylistId` defines the ID of a playlist owned by the authenticated user in which to insert the video lecture, should the user upload it to YouTube using the tool.

4.2 Slide Management

SSML does not support the inclusion of slides of any type because it only focuses on Text-to-Speech generation. However, to render the lecture slides, LSML extends the language with features for referencing PDF documents as slide decks. A slide deck in the context of LSML refers to a set of individual slides combined within a single file.

PDF documents were selected as the file format for slide decks because they are software- and hardware-independent. They can be created in most operating systems without the need to buy a license for specialized software. Additionally, PDF documents are heavily utilized in most universities, including the Bauhaus Universität Weimar.

LSML specifies the empty element `deck` to load a slide deck into the script and assign it a unique identifier. Each page of the PDF document is treated as a slide that can be referenced and loaded in the script, as shown in figure 4.1. The loaded slide will then be rendered as a persistent frame in the video lecture until another frame is loaded.

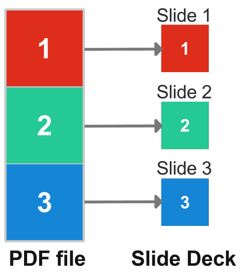


Figure 4.1: PDF Conversion to a Slide Deck

[1] Google Cloud effect profiles: cloud.google.com/text-to-speech/docs/audio-profiles#available_audio_profiles

The `deck` element may appear anywhere in the script, but it must be a direct child of the `lecture` element and must appear at least once. A user may load multiple slide decks, but there must be a `deck` element with a different unique identifier for each one.

The `deck` element specifies a `src` attribute that accepts the path to a local PDF file. A unique identifier is assigned to the slide deck using the `id` attribute. Additionally, exactly one `deck` element must have the attribute `active` set to `true`, which will set the slides deck as the default deck to be used as long as no other slide deck is loaded within a section of or the whole script.

```
<deck id="deck1" src="deck1.pdf" active="true" />
<deck id="deck2" src="deck2.pdf" />
```

By default, a slide loaded from a deck is fitted to the video's output resolution using the `contain` mode shown in figure 4.2. The `contain` mode resizes the slide, so it is fully visible while keeping its aspect ratio. However, the `deck` element implements the attribute `fit` that can change the default scaling mode for all slides loaded from a slide deck. The following three scaling modes exist:

- `contain` resizes the slide, so it fits fully visible inside the frame while keeping its aspect ratio. Areas of the frame not covered by the slide are padded with black pixels. The mode is shown in figure 4.2.
- `cover` resizes the slide to cover the entire frame, while keeping its aspect ratio. If the slide's aspect ratio is different from the output aspect ratio, parts of the slide will not be visible. The mode is shown in figure 4.3.
- `fill` ignores the aspect ratio of the slide to stretch or compress it to fit the output resolution. The mode is shown in figure 4.4.

The following figures demonstrate the scaling modes for a slide with the dimensions of *500x500 pixel* that is resized to fit an output resolution of *1280x720 pixel* using the three different modes.

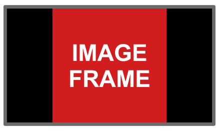


Figure 4.2: Contain Scaling Mode

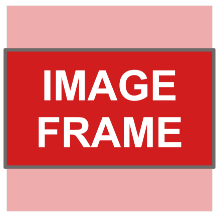


Figure 4.3: Cover Scaling Mode

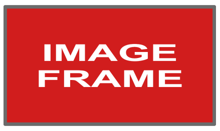


Figure 4.4: Fill Scaling Mode

The usage of the `fit` attribute is simple. It only accepts the names of the three scaling modes as its value: `contain`, `cover` and `fill`.

```
<deck id="deck2" src="deck2.pdf" fit="cover" />
```

A slide from a slide deck can then be referenced in the script using the `slide` element. This element loads the slide into view as a frame that will stay visible until some other slide is loaded. The element has the attribute `deck` that references a slide deck using its ID. If the element does not define this attribute and therefore does not reference any slide deck, the currently active slide deck is used.

The attribute `page` exists to determine which slide to use from the slide deck. It references the slide's page number in the PDF file. The page number can be an absolute value like a non-negative, non-zero integer. The value can also be relative to the currently opened slide by using a signed non-zero integer like *+3* or *-1*. If a defined page number exceeds the available pages in a slide deck, the last page is selected. If it is instead set to below *1*, page *1* is selected. In addition, the attribute also accepts the following keywords:

- `current` corresponds to the current page number.
- `next` loads the next slide in the slide deck.
- `previous` loads the previous slide in the slide deck.
- `first` loads the first slide in the slide deck.
- `last` loads the last slide in the slide deck.

In addition, the `slide` element also specifies a `fit` attribute to set the scaling mode individually for this specific slide. This option will overwrite the default scaling mode set on the `deck` element.

```
<!-- show slide from page 2 -->
<slide page="2" />
<!-- show slide from page 5 -->
<slide page="+3" fit="fill" />
<!-- show slide from last page of a second slide deck -->
<slide deck="deck2" page="last" />
```

4.3 Text Structure

When processing the script, a speech synthesizer's language processor should attempt to determine the paragraph and sentence structure using language-specific knowledge. However, SSML also provides optional elements for explicitly defining the structure of texts should the language processor fail.

These include the elements `p` and `s`, which make the language processor interpret their contents as paragraphs and sentences, respectively.

```
<p>
    <s>This is a sentence in a paragraph.</s>
    <s>This sentence consists of tokens.</s>
</p>
```

The SSML specification also defines tokens, which can be set using the `token` and `w` elements. Both of these elements act the same way and indicate that their content is a token, eliminating word segmentation ambiguities. These elements are necessary for languages that "do not use whitespace for indicating boundaries, for example, Chinese, use whitespace for syllable segmentation, for example, Vietnamese, or use white space for other purposes, for example, Urdu," as stated by the World Wide Web Consortium [2010].

```
Each word is a <w>token</w>.
```

The elements `p` and `s` are supported by both APIs and therefore by LSML as well. Neither API supports the `token` element, and the element `w` is only specified by Amazon Polly. Amazon Polly also accepts the attribute `role` on the element `w` for defining the role of a word, for example, as a verb or a noun. However, this is not supported by LSML, which does not specify any attributes on these elements. Although neither API supports the `token` element, LSML still implements it, and, internally, lecture.js converts it to the to the element `w`. However, because Google Cloud Text-to-Speech does not implement the elements `token` or `w`, they will not have any effect when used in combination with their voices.

4.4 Voices and Languages

Most speech synthesizers support multiple voices and languages, which is why SSML and LSML tailor to that and provide controls for switching voices and languages.

Voices

The `voice` element defines a different narrating voice for its contents. It specifies this voice by a combination of factors. The user can define a desired `gender`, `age`, and one or more `languages`. It is also possible to define a preferred `variant` of speaking characteristics or select a voice by a `name` specific to the language processor. The language processor should consider all the desired characteristics and select the most suitable voice for rendering the element's contents.

```
This sentence is spoken with the default voice.
<voice gender="female" languages="en-GB">
    This sentence is spoken with a different voice.
</voice>
```

Google Cloud Text-to-Speech does not support the `voice` element, and Amazon Polly only implements it with the `name` attribute. In both APIs, selecting a voice is primarily done by explicitly specifying a voice name in the request to the API. LSML therefore also only implements the `voice` element with the `name` attribute, the value of which is then later used in the request to the API. The voice names that lecture.js defines are composed of a prefix corresponding to a Text-to-Speech API, a language code, and the name by which the voice is referred to internally by the corresponding API.

```
<voice name="amazon-en-gb-amy">
    Amy is talking.
</voice>
```

Languages

SSML allows for language changes without changing the active narrating voice using the `lang` element. If words of a different language are used, the language processor should automatically guess their language by their context. However, the `lang` element allows for explicitly defining another language should the language processor fail to detect the language change. The element has a `xml:lang` attribute that accepts ISO language codes with two-letter country codes where applicable. For example, the code `en-US` would set the language to US American English. The element also defines a `onlangfailure` attribute that specifies the language processor's desired behavior upon language speaking failure.

```
German for cat is <lang xml:lang="de">Katze</lang>.
Englisch für Katze ist <lang xml:lang="en">cat</lang>.
```

LSML supports the `lang` element with only the attribute `xml:lang`. Besides, to save the user a little time, LSML defines some similar but shorter language-changing elements. For each language available in the APIs, an element exists with its name being the language code. For example, if British English is available with the language code `en-GB`, the elements `<en-GB>` and `<lang xml:lang="en-GB">` would be synonyms and serve the same purpose. This means, that a user could write the code example from above in a shorter form like this:

```
German for cat is <de>Katze</de>.
Englisch für Katze ist <en>cat</en>.
```

In the SSML specification, the attributes `xml:lang` and `onlangfailure` are permitted on the elements `speak`, `desc`, `p`, `s`, `token` and `w` as well. However, neither of the APIs implements these attributes on those elements, which is the reason why LSML also does not implement them there.

4.5 Correcting Mispronunciations

Speech synthesizers are not always able to produce the correct pronunciation of words, which is why SSML and LSML specify powerful controls for defining pronunciations, one of which is the support for phonetic alphabets. Phonetic alphabets are part of phonetic transcription and are used for precisely transcribing human speech into writing. SSML specifies support for the International Phonetic Alphabet (IPA) and any vendor-specific phonetic alphabet. In practice, however, Text-to-Speech implementations will limit themselves to a small number of alphabets, and some may only implement the IPA.

Lexicons

An important tool to leverage the abilities of phonetic alphabets is the `lexicon` element. It is an empty element that may occur any number of times as an immediate child of the `speak` element in SSML and of the `lecture` element in LSML. Lexicons can define aliases or phonetic pronunciations for specific words.

The `lexicon` element links to a lexicon document using the `uri` attribute. It then gets a unique identifier assigned using the `xml:id` attribute, which can be referenced in specific sections of the document using the `lookup` element. The `lookup` element uses the attribute `ref` to reference the Uniform Resource Identifier (URI) of a lexicon, and the lexicon will then be applied to the contents of the `lookup` element. After a tree of XML nodes is generated from the XML input text, the lexicon that is closest to a text node in the XML tree is the primary lexicon and applied first, while lexicons further away act as a fallback.

```
<lexicon uri="lexicon1.pls" xml:id="lex1"/>
<lexicon uri="lexicon2.pls" xml:id="lex2" />
It does not look up these tokens in any lexicon.
<lookup ref="lex1">
    It looks up these tokens in lexicon 1.
    <lookup ref="lex2">
        It looks up these tokens in lexicon 2.
        If they are not found, it looks them up in lexicon 1.
    </lookup>
</lookup>
```

The `lexicon` and `lookup` elements are not supported by Google Cloud Text-to-Speech, but partly by Amazon Polly. Amazon Polly supports the upload of lexicon files to their cloud console. The lexicon can then be applied to the SSML content by referencing it in the request to the API. However, Amazon Polly does not support dynamically referencing local lexicon files using the `lookup` element.

Because of this, LSML specifies its own variants of the `lexicon` and `lookup` elements, which are preprocessed before the resulting SSML content is sent to the corresponding API. That will enable users to use lexicons for voices of Google Cloud Text-to-Speech, even though they do not support lexicons.

In LSML, the element `lexicon` must only appear as a direct child of the `lecture` element. However, the `lexicon` element no longer references a lexicon file using the `uri` attribute but instead contains the lexicon content itself. Inside, one or more `lexeme` elements can be defined, which are basic lexical units. The `lexeme` element must have exactly two children, one of which must be the `grapheme` element, which defines the word in the SSML content to be replaced. The second element has to be the `alias` element, which defines a replacement word to be pronounced in the grapheme's stead, or the `phoneme` element, which defines a phonetic pronunciation to be used by the narrating voice when pronouncing the grapheme. In contrast, the `lookup` element remains the same as defined in the SSML specification.

```
<lexicon xml:id="lexicon1" alphabet="ipa">
    <lexeme>
        <grapheme>Bob</grapheme>
        <alias>Bobby</alias>
    </lexeme>
    <lexeme>
        <grapheme>tomato</grapheme>
        <phoneme>tə'mei:təʊ</phoneme>
    </lexeme>
</lexicon>
<lookup ref="lexicon1">
    Bob <!-- pronounced as Bobby -->
    tomato <!-- phonetically pronounced as "tə'mei:təʊ" -->
</lookup>
```

Aliases

Lexicons are powerful tools for correcting pronunciation. However, they might be undesirable for smaller tasks, like changing the pronunciation of a single word. For this purpose, SSML and LSML implement elements for accomplishing the tasks of lexicons for individual tokens without using a lexicon.

An example of this is the `sub` element, which allows for the spoken and written form of text in the same document by defining aliases similar to the aliases of lexicons. The element is supported by both APIs, and therefore also by LSML.

The `sub` element defines an alias for the contained text using the `alias` attribute. The alias is the text rendered as speech by the speech synthesizer in place of the text contained within the element. But the element's content remains in the document and can be used for non-audible output like closed captioning.

```
<sub alias="World Wide Web Consortium">W3C</sub>
```

Explicit context

Sometimes, the mispronunciation is not a case of wrongly pronouncing a simple word, but a matter of the language processor guessing the context of a text section incorrectly. In most cases, the pronunciation could be corrected using the `sub` element, but there is also a more elegant solution.

The `say-as` element can be used to provide additional context to a text section as it describes how the text should be interpreted by the language processor. The `interpret-as` attribute indicates the content type of the contained text. It can make the language processor treat the text as a `cardinal` or `ordinal` number, `characters` that need to be spelled out, a `fraction`, an `expletive`, a `unit` of measurement, a `date`, a `time`, a `telephone` number, an `address` or as an `interjection`. It is also possible to provide a specific format, for example, *dmy* for dates, using the `format` attribute.

```
<say-as interpret-as="cardinal">123</say-as>
<!-- One hundred twenty-three -->
<say-as interpret-as="characters">123</say-as>
<!-- One Two Three -->
<say-as interpret-as="ordinal">1.</say-as>
<!-- First -->

<say-as interpret-as="date" format="ymd">2020, May 30</say-as>
<say-as interpret-as="time" format="hms">13:59:59</say-as>
```

The `say-as` element is supported almost in its entirety by both APIs, except for a few interpretation modes. LSML only supports the intersection of interpretation modes available for both APIs, which are: `cardinal`, `ordinal`, `characters`, `spell-out`, `fraction`, `expletive`, `unit`, `date`, `time`, and `telephone`.

Phonemes

The `phoneme` element precisely describes the pronunciation of a small section of text using a phonetic alphabet. In contrast to a lexicon, this element can be comfortably used on smaller, individual text sections. The phonetic string is specified using the `ph` attribute, and the phonetic alphabet can optionally be specified using the `alphabet` attribute.

Though the element is specified in LSML, the `phoneme` element is only supported by Amazon Polly. If it is used together with the voices of Google Cloud Text-to-Speech, it will not have any effect. Amazon Polly also only supports a limited set of characters[2], but provides the implementation of two phonetic alphabets, the IPA and the Extended Speech Assessment Methods Phonetic Alphabet (X-SAMPA).

```
<phoneme alphabet="ipa" ph="pɪˈkɑːn">pecan</phoneme>
<phoneme alphabet="x-sampa" ph='pI"kA:n'>pecan</phoneme>
```

4.6 Prosodic Features

Speech synthesizers automatically determine the prosody to use for the texts they interpret. However, the resulting speech often sounds neutral, which might not fit the speaking style that the user wants to emulate. That is why SSML and LSML implement elements to allow for explicit changes in prosody. Prosody influences the fluency of the speech and how specific sections of text are emphasized.

Emphasis

A simple example is the `emphasis` element, which can instruct the narrating voice to emphasize words or bigger text segments explicitly. The element requests the contained text to be spoken with a specific level of emphasis using the attribute `level`. However, the effect may differ between languages, dialects, voices, and APIs. For example, less emphasis may alter the pronunciation of a phrase like "going to" to "gonna". Both the APIs and LSML support the element in its entirety.

```
It is <emphasis level="strong">strongly</emphasis> emphasized!
```

[2] Supported characters: developer.amazon.com/en-US/docs/alexa/custom-skills/speech-synthesis-markup-language-ssml-reference.html#supported-symbols

Pauses

It is also possible to define a speaking pause in the rendered speech in both SSML and LSML using the `break` element. This pause lasts for a specific duration specified using the `time` attribute. The attribute accepts time units in the time value format from the Cascading Style Sheets Level 2 Recommendation by the World Wide Web Consortium [1998]. These time values consist of a positive number followed by the time unit identifiers `ms` for milliseconds or `s` for seconds.

The pause may also be very abrupt, which can be corrected by setting a lower prosodic strength set using the `strength` attribute.

```
Wait for half a second...
<break time="500ms" strength="medium">
The wait is over.
```

Prosody

The `prosody` element could be described as the *Swiss Army knife* of prosodic controls. Not only does it control the pitch, but also the speaking rate and volume. It specifies the following attributes:

- `pitch` defines the baseline pitch using an absolute Hertz value or a keyword. The pitch may also be set relatively using percentages, semitones, and, once again, Hertz values.
- `range` defines the pitch range, or variability, for the contained text. The meaning of pitch range will vary across language processors, but increasing or decreasing the value will typically increase or decrease the dynamic range of the output pitch, respectively.
- `rate` defines a change in the speaking rate by either speeding up or slowing down the speech.
- `volume` defines the volume for the contained text.
- `duration` defines the desired time it should take for the narrator to read the text. The attribute takes precedence over the `rate` attribute.
- `contour` defines an interpolation of different pitch values as a set of whitespace-separated targets at specified time positions in the speech output.

```
<prosody rate="200%" volume="-20dB">Fast, quiet.</prosody>
<prosody rate="x-slow" volume="x-loud">Slow, loud.</prosody>
<prosody duration="2s">This is spoken in 2 seconds.</prosody>
<prosody contour="(0%,+20%) (50%,+10Hz)">This sentence is
   pronounced in a peculiar way.</prosody>
```

Although LSML supports the `prosody` element, there are some limitations because the APIs have a few peculiarities in how they handle the different attributes. In general, LSML does not restrict the user input for the attributes, and lecture.js sends the values straight to the corresponding API. However, the APIs have different ranges and accepted units for these attributes, all listed in the *LSML Feature Support Table* in the appendix. The user will have to check if their units are supported for the voice which they are using together with the `prosody` element. In addition, the attributes `range`, `duration` and `contour` are not supported by LSML, because of lacking support by the APIs.

4.7 Embedding External Resources

Providing support for external resources makes the markup language and resulting software more versatile. It gives users additional tools to prepare their video lectures besides only relying on slides and markup.

Audio

SSML makes it possible to insert external audio files into the speech output using the `audio` element. The `audio` element implements the following attributes:

- `src` specifies the path or the URL for the media file.
- `clipBegin` defines an offset from the start of the media to begin rendering as a time designation.
- `clipEnd` defines an offset from the start of the media to end rendering as a time designation.
- `soundLevel` defines the relative volume using a positive or negative decibel value.
- `speed` defines the playback speed in percentage using a positive percentage value.
- `repeatCount` defines how often to loop the audio.
- `repeatDur` defines the total duration of repeatedly rendering the media.

The `audio` element can remain empty or contain SSML content. Should the audio file fail to play or the user is generating non-audible output for accessibility reasons, the contents of the `audio` element should be rendered instead.

```
<!-- plays clip twice -->
<audio src="sound.mp3" repeatCount="2" />
<!-- start clip 30 seconds in, ends it at 1 minute -->
<audio src="speech.mp3" clipBegin="30s" clipEnd="60s">
    If the audio file fails to play, this will be read.
</audio>
```

The `desc` element may also only appear inside an `audio` element. Its purpose is to add a description to the audio file if it does not contain audible speech. For example, the element may describe sound effects like a door slamming or a buzzer going off.

```
<audio src="buzzer.wav">
    <desc>Buzzer sound effect</desc>
</audio>
```

However, LSML's implementation of the `audio` element differs in a few ways from the SSML implementation. LSML does not currently support the `desc` element because the subtitles feature is not yet implemented. Therefore, the `audio` element must also remain empty and not contain any other SSML content. While the audio plays, the current frame stays.

The `audio` element is also only partly implemented by Amazon Polly, making it necessary to create a custom implementation of audio insertion into the lecture to avoid sending audio files to the APIs. This custom implementation comes with the following changes to the attributes of the `audio` element:

- `src` only accepts local audio file paths, not URLs.
- Both `clipBegin` and `clipEnd` only support a simplified form of SSML timestamps without floating point numbers, but additionally implement timestamps in the formats *hh:mm:ss* and *hh:mm:ss.SSS*.
- `soundLevel` is limited to integer decibel values between *+50dB* and *-50dB*.
- `speed` is limited to integer percentage values between *50%* and *200%*.
- `repeatDur` is not implemented.

Videos

Because SSML is focused on speech generation, it does not support the insertion of videos. However, video clips can add a lot of interactivity and variety to a video lecture, for example, screen recordings. LSML, therefore, supports the insertion of video clips into the lecture using the empty element `video`. The video loaded by the element will be played in its entirety unless otherwise specified by the user.

The `video` element specifies the attributes `src`, `clipBegin`, `clipEnd`, `soundLevel`, `speed` and `repeatCount` in a similar manner to the LSML `audio` element. In addition, LSML also implements the following attributes for the `video` element:

- `keepFrame` defines if after the video was played, the last frame should remain as the current video lecture frame, or if the lecture should display the previous slide again.
- `fit` defines the scaling mode for fitting the video to the output video resolution as shown in the figures 4.2, 4.3 and 4.4.

```
<!-- plays a video -->
<video src="recording.mp4" />
<!-- plays a video at 2 seconds into the clip -->
<video src="footage.mp4" clipBegin="00:00:02" />
```

Images

Users may want to include a set of images, like screenshots, that are not part of their slides. Instead of forcing them to edit their slides, LSML provides a simple way of embedding an image and setting it as the video presentation's current frame. The empty element `image` supports the direct integration of an image as a frame into the video. The image stays visible until the next `slide`, `video` or `image` element is used. The element implements the following attributes:

- `src` defines the path to a local image file.
- `fit` defines the scaling mode for fitting the image to the output video resolution as shown in the figures 4.2, 4.3 and 4.4.

```
<image src="image.png" fit="cover" />
```

Chapter 5

Implementation

5.1 Development Environment

This section describes the development environment, which includes the programming languages used and the required software. JavaScript of the standard ECMAScript 6 is the programming language that was primarily used to implement lecture.js. JavaScript is particularly useful because it sees widespread usage even outside of web browsers and provides access to many free libraries. Additionally, the usage of JavaScript will make it easier to port the application to a server-side environment to run it as a website in the future, if needed.

Node.js *v12.18.2* was used as the JavaScript runtime environment to realize lecture.js outside the web browser. Node.js is written in *C* and *C++*, which makes the environment run smoothly on Windows, Linux, and macOS environments. In the course of development, lecture.js was successfully tested and deployed on Windows 10 and Ubuntu Linux versions *16.04*,*18.04* and *20.04*. The software may also work on macOS systems, although this was not specifically tested.

Several libraries and modules were used in the creation of the software. "Modules" refer to JavaScript libraries included from *npm*, which is the largest distributor of open-source JavaScript code. In addition, an XSD library was imported for XML validation and the library suite FFmpeg was included to allow for audio and video manipulation. The former requires Java, for which the Java Runtime Environment build *1.8.0_211-b12* was used throughout development. Other technologies were also utilized, including XML markup, XSD schemas for validation, documentation using markdown, and batch and shell scripts for faster execution of basic lecture.js functionality on Windows and Linux operating systems.

A user that wants to use lecture.js will need to install a build of Node.js. However, installing Java is optional as long as the LSML validator is disabled.

5.2 Program Structure

Lecture.js is programmed using a modular structure. Each module consists of a directory with an internal structure and one or more JavaScript files. Each is responsible for one large but specific task. For example, the module *text-to-speech* manages all communication with the Text-to-Speech APIs and the caching of generated audio files. If a script wants to communicate with these APIs, it would need to import the *text-to-speech* module.

Additionally, lecture.js is also organized in a 4-layer structure, as shown in figure 5.1. Each layer may only import and use components from the layers below. The pipeline is the highest layer and the one with which users of the software interact. Although a module in itself, the pipeline conceptually makes up a separate layer, while the other modules make up the second-highest layer. The third-highest layer is the global scripts that may be imported and used by any module. The global scripts are intended to be utility scripts with general functions that one or more modules may need. This makes the whole layer a dependency of any module. The lowest layer comprises the so-called node modules, which are JavaScript libraries included in the software. These libraries may be used by any global script or module, making all of them a dependency of the three higher layers.

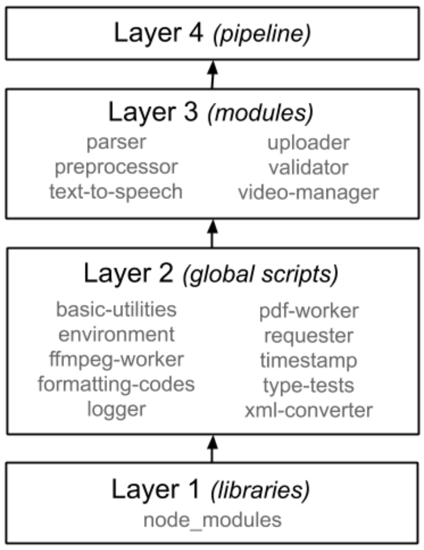


Figure 5.1: 4-Layer Program Structure

The modules were designed with independence in mind, meaning that there should be no dependencies between individual modules. The modules may then be individually extracted from lecture.js and re-used elsewhere like building blocks, except for the dependencies to the lower layers.

Although most of the modules are independent of other modules, that is not possible for some. The parser and preprocessor modules each import the *text-to-speech* module because they need some of its functionality for processing the input script's contents.

Most of the global scripts also import the global scripts *basic-utilities*, *logger* and *type-tests*, making them not completely independent from each other. However, suppose one wants to extract a global script for usage outside the software. In that case, the dependencies are easy to figure out because they are listed on the top of the script, as described in section 5.6. Additionally, imported scripts often only encompass the three above mentioned scripts, making the extraction and re-use rather trivial.

5.3 Modules

5.3.1 Pipeline

The pipeline module is the only part of lecture.js with which the user directly interacts. It is responsible for managing the whole task of video lecture generation from start to end. The pipeline imports all other available modules to use their specific capabilities to generate a video lecture.

Structure

The pipeline module only defines a single public function, which starts the pipeline and generates a video lecture. The pipeline runs until the video lecture is successfully generated or a fatal error is thrown. The program exits the process either using the error code `0` to indicate a successful execution or the error code `1` if a fatal error occurred. Only the pipeline module may throw fatal errors and exit the program. Other modules are only allowed to trigger basic errors and warnings and then return a status code, `undefined` or `false`. The invoking module should then act similarly. Eventually, the error should be passed back to the main executing module, the pipeline. The pipeline should then handle the error and possibly exit the program with a fatal error message. Additionally, modules besides the pipeline should print no unnecessary status logs to the terminal.

After the pipeline is started, it parses the user's command-line arguments[1] to determine which mode to use. A configuration file is parsed to determine critical run-time options.[2]

[1] Available command-line arguments are explained in section "Terminal Access."
[2] Available configuration options are explained in section "Configuration."

The pipeline has three different modes it may use upon execution:

- **Information Mode**: If the user requests some information, for example, the help menu or a list of available voices, the pipeline prints the requested information and exits.
- **Sample Mode**: If the user requests to generate a Text-to-Speech audio sample, the pipeline uses the provided text content and arguments to generate an MP3 audio file and exits.
- **Lecture Mode**: If the user requests to generate a video lecture, the pipeline uses the provided input script to generate a video lecture in the defined output directory and exits.

If the *Lecture Mode* is run, the pipeline uses the validator module to validate the input script. If the input script is valid LSML markup, the pipeline will attempt to parse the meta-information from the input script using the parser module. Once that is completed, the preprocessor module is called, and the input script is preprocessed to optimize it for the later sections of the pipeline. The preprocessed script is then parsed using the parser module and converted to a JavaScript Object Notation (JSON) representation, where the script is split into smaller sections. Each section corresponds to a section of SSML that is later sent to a Text-to-Speech API to be converted into speech. The parser then combines the resulting JSON data with the parsed meta-information and transforms it into the pipeline's internal data format. The subsequent parts of the pipeline all work on this internal data object and extend it if needed.

Once the internal data object is created, the pipeline creates the output directory structure. Depending on the command-line arguments, the files and folders may be directly created inside the defined output directory, as shown in figure 5.2 or a subdirectory will be created in which all of them are placed, shown in figure 5.3. A directory is generated for each type of video component so that they may be looked at individually or re-used later.

```
output/
├── audio/
├── clips/
└── frames/
```

Figure 5.2: Direct Output Directory Structure

```
output/
└── 20201014064516448-example/
    ├── audio/
    ├── clips/
    └── frames/
```

Figure 5.3: Wrapped Output Directory Structure

The direct output directory structure is preferable if lecture.js is called from another program because it removes the aspect of the unpredictable name for the output directory. However, the wrapped output directory structure is more useful for human usage because multiple lectures can be generated in the same output directory and are neatly sorted by date and name.

After the output directory is created, any video and audio clips embedded into the script using the `video` and `audio` markup elements are converted to the right codec and then placed in the output directory. Additionally, they may be trimmed and have their volume or speed changed, depending on the user's settings. The pipeline then creates the frames for the video lectures and converts the SSML sections into Text-to-Speech audio files. Both are then combined to create an individual video clip for each section. These clips are later concatenated to the final video lecture. The user then has the option to upload the video lecture to YouTube if a credentials file for the YouTube Data API is defined in the configuration file.

Configuration

The pipeline module includes a configuration file of the `.ini` file format. Several options related to the execution of the pipeline are defined within the file. All settings that can also be changed using a `settings` element for an individual input script have a corresponding default value in the configuration file, which is used if the setting is not defined in the script. The path to the credentials files needed for the Amazon Polly, Google Cloud Text-to-Speech, and YouTube APIs are also defined here either as relative or absolute paths. Because the credentials files are optional as long as at least one valid credentials file for a Text-to-Speech API is provided, not all of the APIs have to be used. This makes the installation more time-efficient if the user only needs the voices of one API. If the credentials for YouTube's Data API are not provided, only the direct upload to YouTube will be affected and not work. Additionally, the configuration offers options for quite a few other aspects of the software, especially for caching[3]:

- `generic.enableValidator` defines if the input script should be validated.
- `generic.outputData` defines if the internal data structure that was generated from the input script should be outputted as a JSON file.
- `generic.outputLogFile` defines if a log file should be created.
- `generic.outputFFmpegLogs` defines if an FFmpeg log file should be created for every significant FFmpeg operation used at run-time.

[3] Caching is explained in section 5.3.5.

- `log.colored` defines if logs should be printed in color to the terminal.
- `log.maxCount` defines the maximum number of log entries per log file.
- `cache.directory` defines the absolute or relative path to the directory where cached files should be placed.
- `cache.defaultMode` defines the default cache mode:
  - `off` : No caching is used.
  - `on` : Audio files can be loaded from the cache, and newly generated audio files can be saved in the cache.
  - `readonly` : Audio files can be loaded from the cache, but newly generated audio files will not be saved in the cache.
  - `saveonly` : Audio files can not be loaded from the cache, but newly generated audio files will be saved in the cache.
- `cache.expiresInDays` defines how many days after an audio file is cached, it is removed from the cache.
- `upload.generateYoutubeDescription` defines if a YouTube description should be generated as a text file for every output.

Terminal Access

When the main lecture.js JavaScript file is executed from the terminal, it calls only the pipeline module, which means that the pipeline module is responsible for processing any command-line arguments supplied to the lecture.js script. Consequently, the pipeline can be run using the computer terminal and be integrated into other programs. For this human and programmatic usage, the pipeline defines an interface consisting of different flags and parameters. First of all, there are some flags for obtaining meta-information about the program:

- `-h` and `--help` print the help menu.
- `-v` and `--version` print the version of the program.
- `--voices` prints a list of available voices and their parameters.
- `--languages` prints a list of available language codes.

The user can generate any video lecture with additional options, but a basic command only needs an input script and an output directory defined. However, all these parameters and flags are available:

- `-i` or `--input` defines the relative or absolute path to the input script.

- `-o` or `--output` defines the relative or absolute path to the output directory.
- `--nowrap` tells the pipeline to output files directly into the output directory. If this flag is not set, a subdirectory will be created inside the given output directory, which will contain all the output files instead.
- `--cache` defines the cache mode to use for any Text-to-Speech requests.

To generate the example lecture script provided in the project, a user would need to type the following command into their terminal:

```
node lecture.js -i "input/example.xml" -o "output/"
```

The cache may also be cleared using the flag `--clearcache`. If the flag is used, all cached audio files are removed.

Additionally, the pipeline also supports the generation of audio samples with a chosen voice without running the whole pipeline. This is intended for the user to be able to try out certain voices and SSML features. The pipeline implements the following parameters and flags for the generation of audio samples:

- `-s` or `--sample` tells the pipeline that a sample should be generated.
- `--voice` defines the name of the voice to render the sample text.
- `--text` defines the sample text to be rendered as audio.
- `--ssml` tells the pipeline to render the text as SSML content and not as plain text.

To generate a small audio sample, the user could use the following command in their terminal:

```
node lecture.js -s --voice "amazon-en-gb-amy" --text "Hello!"
```

5.3.2 Validator

SSML is based on XML, and consequently, LSML, as an extension of SSML, is also XML-based. XML is easy to validate, because well-proven technologies to do just that already exist, namely XSD schemas, which are a formal way to describe all the elements and their attributes in an XML document. The validator module uses the library xjparse[4], which is a wrapper of the open-source XML processor Apache Xerces[5], to validate XML files using XSD schemas. The validator may be called with the path to any XML file to validate it using a custom LSML XSD schema.

[4] xjparse: github.com/ndw/xjparse
[5] Apache Xerces: xerces.apache.org

For SSML, there already exists an XSD schema provided by the World Wide Web Consortium [2010]. Lecture.js imports this schema in its own custom schema definition for LSML and re-uses all the SSML elements that are not different from the LSML specification. The other elements are re-defined in the LSML schema, and the SSML elements not defined in LSML were omitted. Additionally, LSML extends the schema with the definitions for its own custom elements, which do not exist in the SSML specification. A copy of the LSML XSD schema is attached in the appendix, starting on page V.

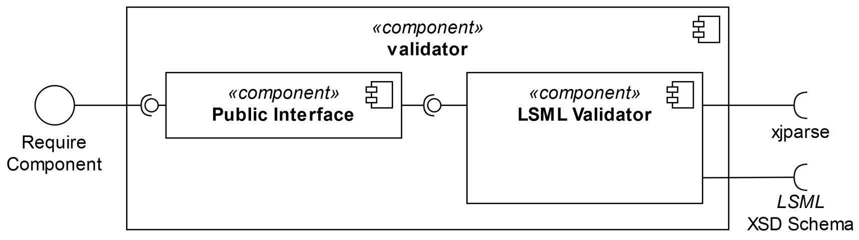


Figure 5.4: Validator Module UML Component Diagram

5.3.3 Preprocessor

The preprocessor module takes an LSML document and transforms its contents, and applies specific changes to make it fit for usage in the later stages of the pipeline. It aims to remove any elements that can not be processed by the validator module, which is the case if any information requested from the Text-to-Speech APIs is required to validate an element. This information is dynamically requested at run-time from the APIs, making it impossible to define it statically in an XSD schema. Additionally, the preprocessor converts and directly applies elements to the script that are not or only partially defined in the Text-to-Speech APIs. The preprocessor performs the following operations on the LSML script:

1. It adds pauses of a defined duration between paragraphs by inserting `break` markup elements.
2. It adds a pause of a defined duration whenever a slide changes by inserting a `break` markup element.
3. It transforms any `token` markup element into a `w` markup element because Amazon Polly does not support the former, but the latter.
4. It removes `voice` markup elements that define a non-existing voice.
5. It removes `lang` markup elements that define a non-existing language.

6. It converts `lang` markup elements defined in a section where a voice from Google Cloud Text-to-Speech is active to a `voice` markup element with the name of a similar voice in a different language. This is only done for Google Cloud Text-to-Speech because it does not support the `lang` markup element.
7. It transforms any specific language tags defined by LSML to an SSML equivalent. For example, the markup elements `de` or `en-US` would be converted to a corresponding `lang` markup element if a voice from Amazon Polly is active, or to a `voice` markup element with a similar voice in the corresponding language if a voice from Google Cloud Text-to-Speech is active.
8. It applies any defined `lexicon` markup element wherever it is referenced using a `lookup` markup element. The affected text sections will be scanned for the occurrences of graphemes defined in the corresponding lexicons. If one is found, it is wrapped with either a `sub` markup element if the lexicon defines an alias or with a `phoneme` markup element if the lexicon defines a phonetic pronunciation. After all lexicons are applied, all `lexicon` and `lookup` markup elements are removed.

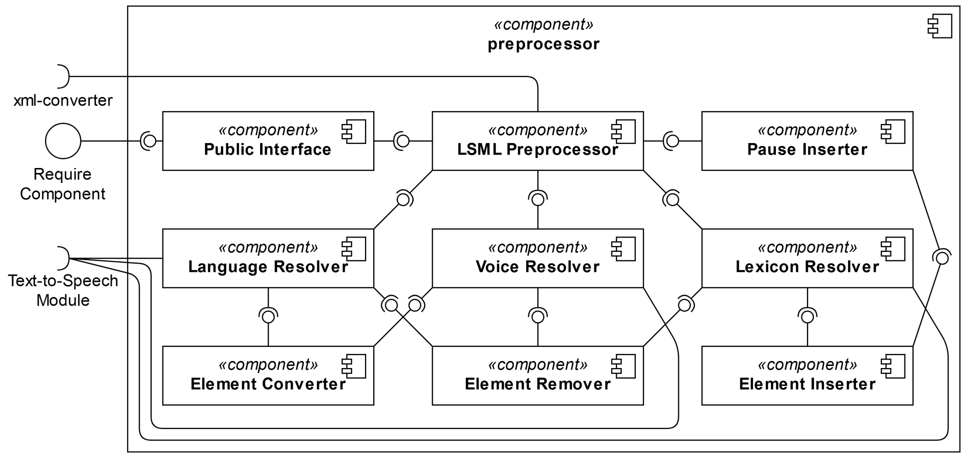

«component»
preprocessor
xml-converter
Require Component
Text-to-Speech Module
«component» Public Interface
«component» LSML Preprocessor
«component» Pause Inserter
«component» Language Resolver
«component» Voice Resolver
«component» Lexicon Resolver
«component» Element Converter
«component» Element Remover
«component» Element Inserter

Figure 5.5: Preprocessor Module UML Component Diagram

5.3.4 Parser

The parser module is responsible for parsing the input script using LSML-specific knowledge and generating an internal data object that will be used and extended by the pipeline. The module has two primary sub-components, which may be called upon by external modules.

The first sub-component of the parser is the meta parser, which is responsible for executing a shallow parse of LSML content and extracting meta-information. This meta-information includes all `info`, `settings`, `deck`, and `lexicon` markup elements, which are all direct children of the `lecture` root element. Because LSML is XML-based, an existing XML-parser can be used to parse LSML content. Lecture.js imports the library *xml-js*[6], and implements the global script *xml-converter* as an interface for the library with which other modules and global scripts may communicate. The meta parser calls the *xml-converter* to convert the LSML content to a JSON representation, as shown below.

```
{
    "elements": [
        {
            "type": "element",
            "name": "lecture",
            "attributes": {
                "startmark": "intro"
            },
            "elements": [
                ...
            ]
        }
    ]
}
```

This JSON representation can then be analyzed for meta-information, which is converted to the internal data format and results in output similar to what is shown below. Additionally, the pipeline module will generate a Universally Unique Identifier (UUID) for the data object and add the date of when the script was parsed. The data generated by the meta parser may include settings that alter the preprocessor module's behavior, like the `breakAfterSlide` setting. The pipeline must therefore call the meta parser before the preprocessor transforms the input script.

```
{
    "info": {
        "title": "Simple Example"
    },
    "settings": {
        "voice": "amazon-en-gb-amy",
        "resolution": {
            "width": 1280,
            "height": 720
        },
        "fps": 30,
        "breakAfterSlide" : 1300
    },
    "decks": {
        "intro-slides": {
            "id": "intro-slides",
            "file": "intro.pdf",
```

6 xml-js: npmjs.com/package/xml-js

```
            "file_path": "path\\to\\intro.pdf",
            "pages": 10,
            "fit": "contain"
        }
    },
    "active_deck": "intro-slides",
    "lexicons": [
        {
            "id": "lexicon1",
            "alphabet": "ipa",
            "lexemes": [
                {
                    "grapheme": "Bob",
                    "type": "alias",
                    "replacement": "Bobby
                },
                {
                    "grapheme": "tomato",
                    "type": "phoneme",
                    "replacement": "təˈmeiːtəʊ"
                }
            ]
        }
    ],
    "uuid": "2cf70fd4-61ff-4af8-abe3-6b86b7cf3532",
    "date": "2020/10/14 06:45:16.448"
}
```

The second sub-component is the section creator, which is called after the preprocessor module is executed. The section creator runs a deep parse of the whole preprocessed LSML content. One of the module's tasks is to scan for any marks, chapters, and embedded media resources to create additional meta-information entries. This results in data similar to what is shown below.

```
{
    "resources": {
        "video": [
            {
                "id": 0,
                "path": "path\\to\\video.mp4",
                "variants": {
                    "START-00:00:03.500": "path\\to\\trimmed\\video.mp4"
                }
            }
        ],
        "image": [
            {
                "id": 0,
                "path": "path\\to\\image.png"
            }
        ],
        "audio": [
            {
                "id": 0,
                "path": "path\\to\\audio.mp3",
                "variants": {
                    "START-00:00:02.000": "path\\to\\trimmed\\audio.mp3"
                }
            }
        ]
    },
    "marks": {
        "startmark": "intro",
        "content": {
            "intro": {
                "section": 1
            },
            "toc": {
                "section": 2
            }
        }
    },
    "chapters": [
        {
            "title": "Intro",
            "mark": "intro",
            "section": 1
        },
        {
            "title": "Table of Contents",
            "mark": "toc",
            "section": 2
        }
    ]
}
```

However, the section creator's primary purpose is to split the input script into sections at specific positions. Each section is later converted to speech separately from the others, meaning that each section corresponds to an individual request sent to a Text-to-Speech API. Consequently, a new section must be created whenever an element is encountered, which can not be directly handled in the request body that contains the SSML content, but only in the request head. This is the case for the `voice` markup element. Additionally, a new section must be created whenever an element is invoked, for which the pipeline needs to know the start timestamp. Because the Text-to-Speech APIs do not return timestamps for words and sentences, the pipeline can only determine timestamps beforehand by splitting the content into sections which are later converted to individual video clips and concatenated. The pipeline needs to know the timestamp whenever a new chapter is created, a media file is inserted, or a frame is changed. This means that a new section is created whenever a `mark`, `video`, `image`, `audio`, or `slide` markup element is encountered.

```
{
    "sections": [
        {
            "id": 1,
            "type": "ssml",
            "frame": {
                "type": "slide",
                "deck": "intro-slides",
                "page": 1,
                "fit": "contain"
            },
            "voice": "amazon-en-gb-amy",
            "content": "<speak>\r\n\r\nGood day!\r\n\r\n</speak>"
        },

        ...

    ]
}
```

Additionally, a warning is printed once a section becomes too large. Amazon Polly and Google Cloud Text-to-Speech both set limits to the number of characters that may be sent in a single request to their APIs. Amazon Polly currently allows for 3000 billed plain text characters and an additional 3000 non-billed SSML characters per request, while Google Cloud Text-to-Speech limits its requests to 5000 characters in total. The section creator prints a warning once a section exceeds the limit set for the associated API. However, it does not automatically split the section because this would require a reliable detection for the text's sentence and paragraph structure, which was too time-consuming to implement.

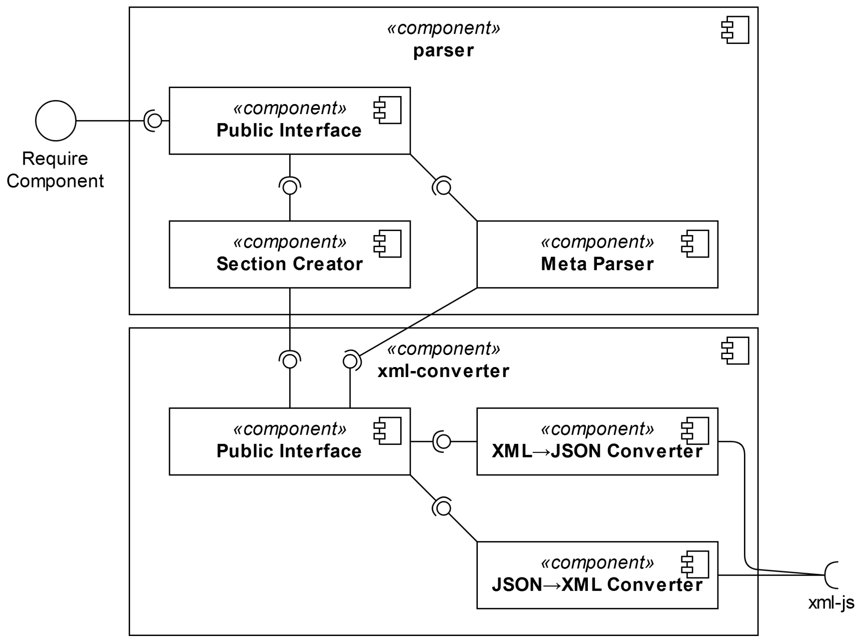


Figure 5.6: Parser Module UML Component Diagram

5.3.5 Text-to-Speech

The two Text-to-Speech APIs Amazon Polly and Google Cloud Text-to-Speech were integrated into lecture.js. The latter has excellent voices that sound very natural, but it lacks in terms of SSML support. Google has been slow in adopting new features since the introduction of its Text-to-Speech service, which might make their voices much less valuable in the future.

On the opposite end, Amazon Polly supports most of the essential SSML features, but their voices, except their new neural voices, have an audibly worse sound and tend to stutter every so often. However, Amazon already announced the Neural Text-to-Speech system on their AWS Machine Learning Blog [2019], presenting higher quality neural voices for the languages English, Portuguese, and Spanish. Amazon Polly's neural voices still exhibit some of the same problems as the standard voices but sound audibly better. If they improve their voices even further and approach Amazon Alexa in terms of voice quality, it may become the go-to source for voices in lecture.js.

API Integration

The *text-to-speech* module was created to integrate the two above-mentioned APIs into lecture.js. The module aims to standardize Text-to-Speech APIs in lecture.js by acting as a common interface between the modules and the different APIs. Consequently, this also makes it easier to integrate additional Text-to-Speech services into lecture.js in the future.

The *text-to-speech* module provides several standardized functions for getting information about the currently available voices and languages from the APIs, including the validation of voice names and language codes. Additionally, it provides functions to find similar voices and language codes to a given key. For example, the module can find a similar voice to a male US-American English voice in another language. These features are used by other modules, like the parser and preprocessor, and the pipeline, to validate and process any user input related to the Text-to-Speech APIs.

Any module outside the *text-to-speech* module does not need to know to which API a voice belongs because the module internally determines the correct API to use depending on the parameters the user provides. The module can render both SSML and plain text as speech. It defines the following parameters for a Text-to-Speech request:

- `voice_name` specifies the name of the voice specific to lecture.js. This name is different from the voice name set by the corresponding API to prevent clashes between similar voice names across APIs.
- `type` specifies the type of content to render, which may be either SSML or plain text.
- `content` defines the text to be rendered as speech.

Once the first request is sent to the module, it establishes a connection to Amazon Polly and Google Cloud Text-to-Speech using the credentials files referenced in the configuration file.

Because the credentials files are optional as long as at least one valid credentials file is provided, there may be cases where only one connection to an API is established. If the request is a speech synthesis request, the module determines the API corresponding to the `voice_name`, and then an object with the pertinent data for the API is created and sent to the API. The API, in turn, sends an HTTPS request to a web server, where the audio is being generated. After a while, a binary string of MP3 audio data is returned by the API and sent back to the module that invoked the *text-to-speech* module.

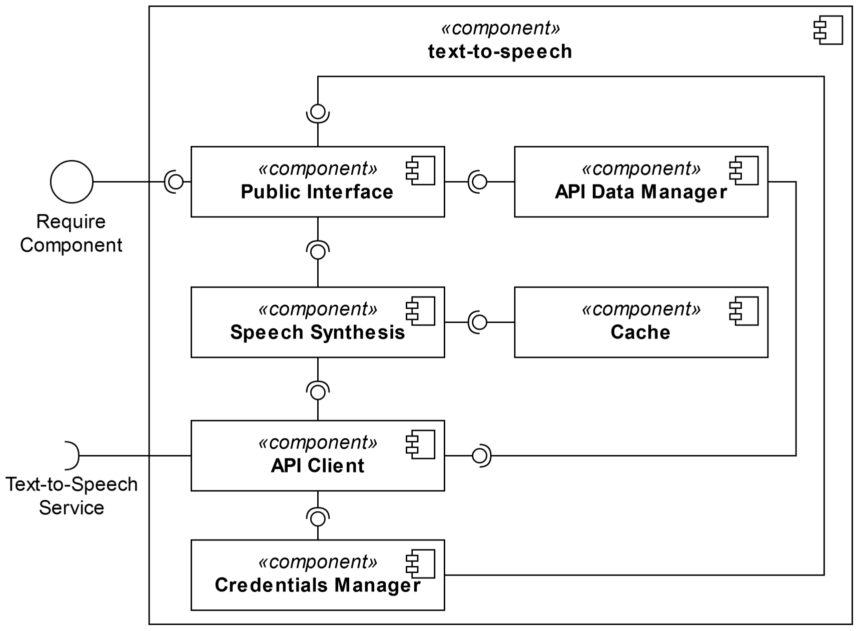


Figure 5.7: Text-to-Speech Module UML Component Diagram

Caching

Sending requests to the Text-to-Speech APIs requires both time and money, which does not scale well for an application that might be used repeatedly by multiple users. Each request to the APIs is a request to a server that must be resolved before the program can continue. These services offer free quotas for how many characters may be requested per month. However, after the limits are reached, the user must usually pay around *$4* for standard voices and *$16* for higher quality voices per 1 million characters. Because the standard voices sound very robotic, the higher quality voices, which are the WaveNet voices of Google Cloud and neural voices of Amazon Polly, will probably be used exclusively by the users. These accrued costs may grow exceptionally fast if a user is repeatedly generating the same lecture with only minimal changes to the input text to correct individual words or phrases.

In such a case, the program would repeatedly request the two Text-to-Speech APIs for the same full script. Caching can remedy this situation by saving previously generated audio files temporarily to be re-used should the same text be requested again.

A caching manager was added to the *text-to-speech* module to control a simple cache. Whenever the module receives a speech synthesis request for the APIs, an ID will be generated that uniquely identifies the request. The ID consists of the voice's name, a Google Cloud Effect profile if applicable, the type of content, either SSML or plain text, and the actual content to be rendered as speech. Because the resulting ID might be very long, a compression algorithm[7] based on Lempel-Ziv is used to compress the ID in the UTF16 character encoding to around *50%* of the original string's length. The ID is then matched against a simple lookup-table implemented using JSON. If the ID exists in the table, the binary data for the audio file is returned. Otherwise, the speech synthesis function is executed, and the resulting audio file will be cached using the generated ID.

The cache database itself consists of a directory placed in the project's root directory that contains all the cached audio files and a JSON file that indexes the IDs with the corresponding audio file paths and the date of entry. The index file is read once the cache is accessed for the first time, and a self-reparation process is run to keep the cache consistent. All audio files that are cached but not indexed are removed. In addition, all entries in the index file for which no corresponding audio file exists are removed from the index file. Self-repair is not essential unless the program is stopped during execution or the user manipulated the cache. It is also important to remove cached entries that are very old from the cache to prevent the user from re-using the same audio files forever. By default, any audio file older than 365 days is removed. This is necessary because the Text-to-Speech APIs may have changed their implementation in the meantime, which would make a voice used in the lecture sound different depending on if the audio comes from the cache or is newly generated.

Most aspects of how the cache behaves, including the expiry date for cached audio files, may be set in the configuration file or using command-line arguments, as explained in section 5.3.1.

5.3.6 Frame Extraction

Frame extraction in lecture.js refers to extracting an image from a media resource or slide deck and rendering it as a frame in the video lecture.

[7] lz-string: github.com/pieroxy/lz-string

There is no singular module for frame extraction in lecture.js. Instead, the process is realized using multiple global scripts and libraries for different frame types. There are three frame types defined in lecture.js:

1. **Image**: An image is inserted using the `image` markup element.
2. **Slide**: A page of a PDF document is extracted as an image.
3. **Persistent Video Frame**: The last frame of a video resource that was inserted using the `video` markup element is extracted and stays after the video stops playing.

Image

The aim of frame extraction is for an image to be generated. Because the `image` element already defines the path to an existing image file, there is nothing to be done by the pipeline besides noting the file path in the internal data object.

Slide

PDF documents are often utilized in university lectures as slides, because they are shareable documents of images and formatted text. This also makes the format a suitable choice for the type of slides to use in lecture.js. To work with PDF documents and extract the pages referenced in the script as a video frame, lecture.js imports the library PDF.js[8]. This library is a popular PDF viewer and processor developed by Mozilla, being utilized in a wide range of projects, including the browser Mozilla Firefox. The library produced superior results compared to the popular image manipulation library GraphicsMagick, which was also tested in development. PDF.js can also extract additional meta-information from PDF documents required in lecture.js, like counting how many pages the document contains. However, a disadvantage of the library is that it was designed for usage in a browser. Although a Node.js build[9] of the library exists, some of its features do not properly work outside the browser environment. In a web browser, a `canvas` element is created by PDF.js, and the image data is written into it when a PDF document gets converted to an image. This is why it is necessary to simulate the JavaScript Canvas API implemented in web browsers inside lecture.js. For this purpose, the library *canvas*[10] was imported to simulate a HTML `canvas` element inside the Node.js environment.

[8] PDF.js: mozilla.github.io/pdf.js
[9] Node.js build of PDF.js: npmjs.com/package/pdfjs-dist
[10] canvas: npmjs.com/package/canvas

The global script *pdf-worker* was created to provide a common interface between the libraries mentioned above and other modules. The *pdf-worker* script defines a *CanvasFactory* that utilizes the *canvas* library. Once a request for a frame extraction is sent with the file path to a PDF document, a page number, and the desired image resolution, an instance of the *CanvasFactory* is instantiated. This instance can be used to simulate a new `canvas` element. A request with the parameters and the canvas instance is then sent to the PDF.js library. A viewport is calculated to fit the PDF page into the desired resolution, and an image is rendered depicting the PDF page. Because the process uses the library PDF.js, which is also utilized in Mozilla Firefox, the user may check beforehand if their PDF document will be rendered correctly by opening it in Mozilla Firefox and checking for any artifacts.

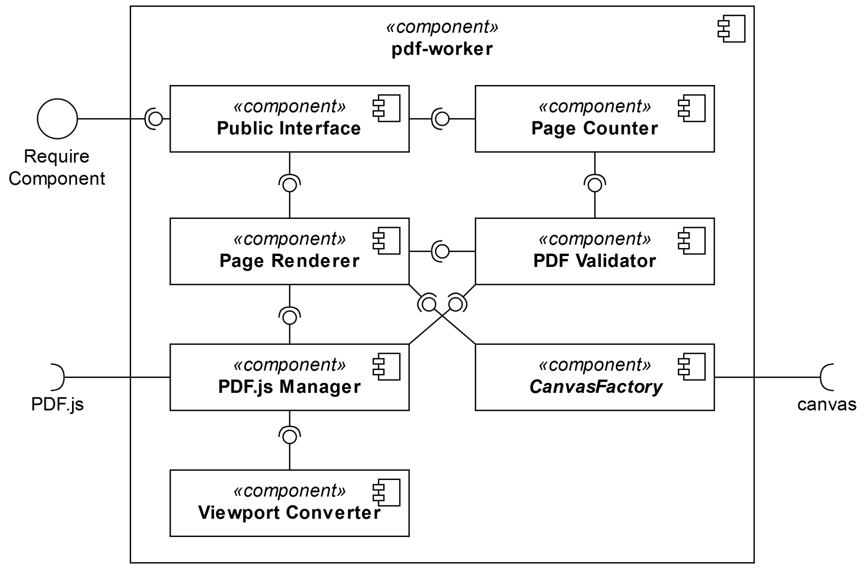

«component»
pdf-worker
Require Component
«component» Public Interface
«component» Page Counter
«component» Page Renderer
«component» PDF Validator
«component» PDF.js Manager
«component» CanvasFactory
«component» Viewport Converter
PDF.js
canvas

Figure 5.8: PDF Worker Module UML Component Diagram

Persistent Video Frame

The open-source software FFmpeg[11] was imported into lecture.js to handle the creation and manipulation of video and audio files. The global script *ffmpeg-worker* was created to act as an interface between FFmpeg and the lecture.js modules. The script implements a function for extracting the last frame of a video file as an image, which is utilized for generating the persistent video frame. The pipeline calls the function with the file path to the converted video resource, places the then generated image file in the output directory, and saves the file path in the internal data object.

[11] FFmpeg: ffmpeg.org

A simplified UML component diagram of the *ffmpeg-worker* is shown in figure 5.9. The usage of FFmpeg in lecture.js and the *ffmpeg-worker* component are explained in more detail in the section 5.3.7.

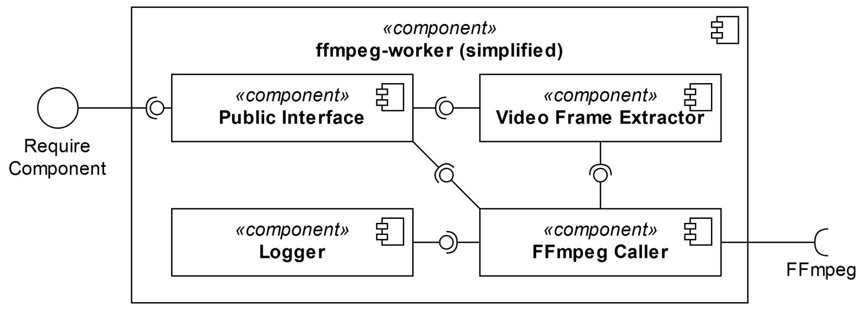


Figure 5.9: Simplified FFmpeg Worker Module UML Component Diagram

5.3.7 Video Generation

Similarly to frame extraction, video generation in lecture.js is not realized using a singular module but a combination of modules, global scripts, and libraries. The manipulation and creation of video and audio files happen at multiple points throughout the lecture.js pipeline. To power these complex tasks, the popular software project FFmpeg was imported into lecture.js. To be more precise, lecture.js utilizes the command-line tools *ffmpeg* and *ffprobe* to analyze and transcribe multimedia files, respectively. To give an efficient way of accessing these tools, the global script *ffmpeg-worker* was created as an interface. The script is utilized extensively by the pipeline to convert the embedded resources to the right codec and apply the user parameters. It is also used to combine audio files and frames to video clips, which are later concatenated to the video lecture. In total, the script provides the following abstracted sub-components:

- The **FFmpeg Caller** and **FFprobe Caller** sub-components allow for generic command-line access to the *ffmpeg* and *ffprobe* command-line tools using any of the available command-line arguments.
- The **Stream Analyzer** can extract information from audio or video streams, which includes the video resolution, the total number of frames, and the duration.
- The **MP4 Creator** combines an audio and image file to an MP4 video file. It accepts several user parameters, including the resolution, scaling mode, and frames per second.
- The **MP4 Concatenator** combines a sequence of MP4 video files to a single MP4 video file.

- The **Video Converter** converts a video to the MP4 container with the video codec *libx264*, the pixel format *YUV420p* and the audio codec *aac*. These encoding settings are recommended by YouTube[12] for uploaded videos, the API of which will need to be integrated into lecture.js. Additionally, it scales the video using the desired parameters and renders the video with the desired frames per second.
- The **Video Trimmer** trims a video to the desired start and end time.
- The **Volume Changer** changes the volume of an audio or video file using a relative decibel amount between *-50dB* and *+50dB*.
- The **Speed Changer** changes the speed of an audio or video file by an absolute percentage value between *50%* and *200%*. Therefore, it can at most halve or double the speed and duration of the media.
- The **Video Frame Extractor** renders the last frame of a video as an image file.

Whenever one of these sub-components is called, an FFmpeg log file can be created alongside the output file unless it is deactivated in the configuration file. The log file might give clues to potential problems if the user encountered an FFmpeg error while using the *ffmpeg-worker*. The pipeline also uses the stream analyzer sub-component to determine the start and end timestamps of each video section, which is later used to generate a table of contents.

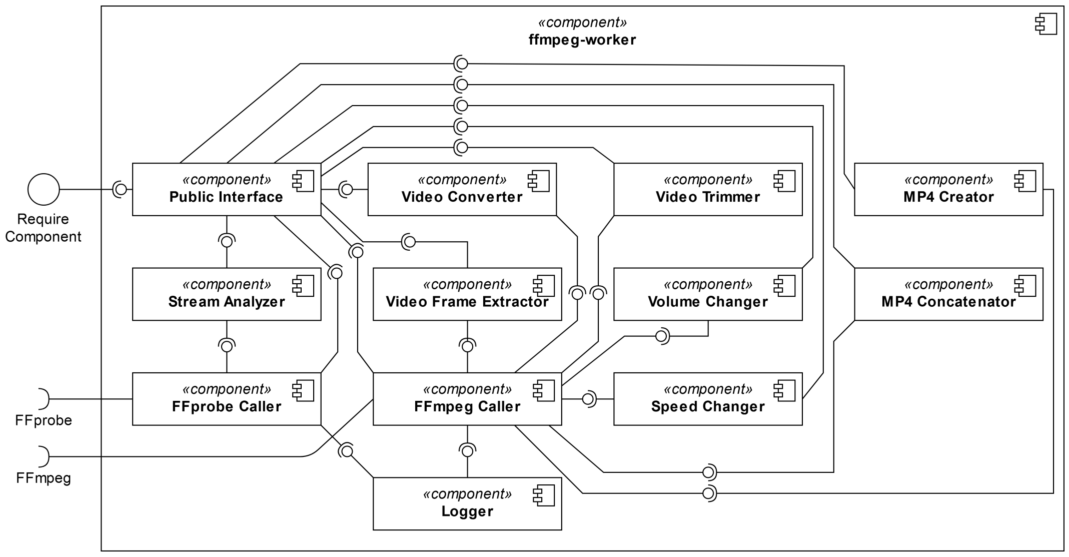


Figure 5.10: Complete FFmpeg Worker Module UML Component Diagram

[12] YouTube's recommended upload encoding settings: support.google.com/youtube/answer/1722171

Additionally, another simple module exists to define the upper and lower boundaries for video parameters of any video generated using lecture.js. This *video-manager* module keeps the parameters consistent across the project except for the validator, which also defines the value statically inside the XSD schema. The *video-manager* can validate the video parameters, as well as, parse LSML video resolution strings in the form of *"{width}x{height}"*.

The module currently allows for videos to have *4k* resolution and up to 120 frames per second. A resolution or frame-rate higher than those values will not result in substantial visible quality improvements because *Full HD* resolution and 30-60 FPS video playback are the norm for most online users. The improvement in video quality will not justify the exponentially growing file sizes and upload times. However, the module also sets lower boundaries because if these parameters were to be set too small by the user, FFmpeg might crash. The *video-manager* defines the following boundaries:

- `UPPER_LIMIT_FPS` = 120
- `LOWER_LIMIT_FPS` = 10
- `UPPER_LIMIT_WIDTH` = 3840
- `LOWER_LIMIT_WIDTH` = 128
- `UPPER_LIMIT_HEIGHT` = 2160
- `LOWER_LIMIT_HEIGHT` = 72

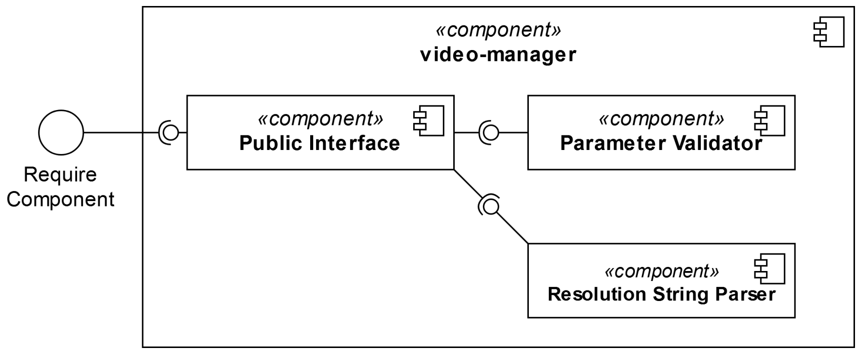
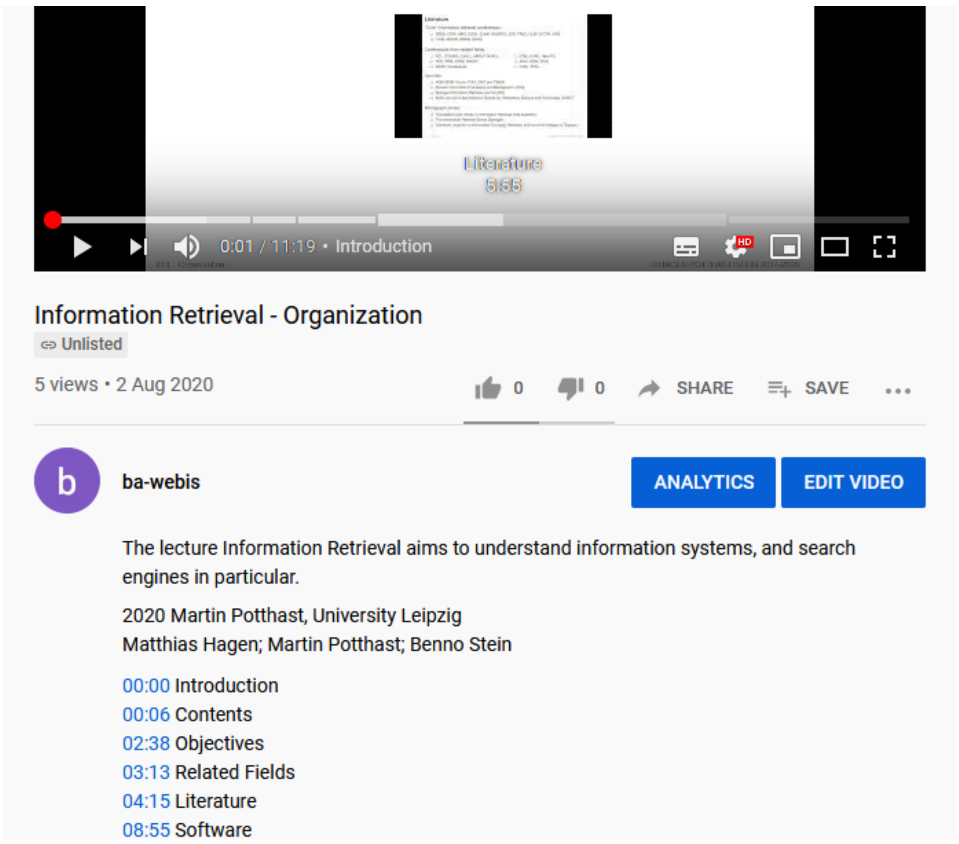


Figure 5.11: Video Manager Module UML Component Diagram

5.3.8 Uploader

The uploader module was added as a supplemental feature to automate the upload process of the generated video lectures to the internet. It is intended to be extended in the future to support multiple web video platforms. For instance, the e-learning platform Moodle would be a welcome addition. At this time, the uploader only supports video uploads to YouTube.

Once the pipeline has successfully generated a video lecture, it will create a video description suitable for YouTube using a function provided by the uploader module. This process happens by default, even if the user does not choose to upload the video afterwards, but this can also be disabled in the configuration file. The description consists of the formatted meta-information that the user defined using the `info` markup element. Additionally, if any chapters were defined using the `mark` markup element inside the script, the module uses them to generate a table of contents inside the description. YouTube uses these timestamps to indicate in their video player where specific chapters of the video begin and end, as shown in figure 5.12.



The lecture Information Retrieval aims to understand information systems, and search engines in particular.

2020 Martin Potthast, University Leipzig
Matthias Hagen; Martin Potthast; Benno Stein

00:00 Introduction
00:06 Contents
02:38 Objectives
03:13 Related Fields
04:15 Literature
08:55 Software

Figure 5.12: YouTube's Video Player with Chapters

After the description is outputted as a text file, the pipeline checks if a credentials file for the YouTube Data API is defined in the configuration file. In case a valid credentials file exists, the user is asked for confirmation to upload the video. If the user accepts the request, an authentication link is printed in the terminal. If the user uses Windows 10, the link also gets directly opened in their default browser. The web page shows a Google authentication prompt, and the user must log in with their YouTube account details. Once they successfully logged in, a token is generated, which the user will need to paste back into the terminal to start the upload process. Once that is accomplished, the uploader module is requested to upload the video lecture. The process of authentication is only required once, and the generated token will be saved to speed up the upload process for future generated lectures to the account. Additionally, the module also supports the insertion of the uploaded videos into playlists owned by the authenticated user. The ID for this playlist may be set in the configuration file or the `settings` markup element.

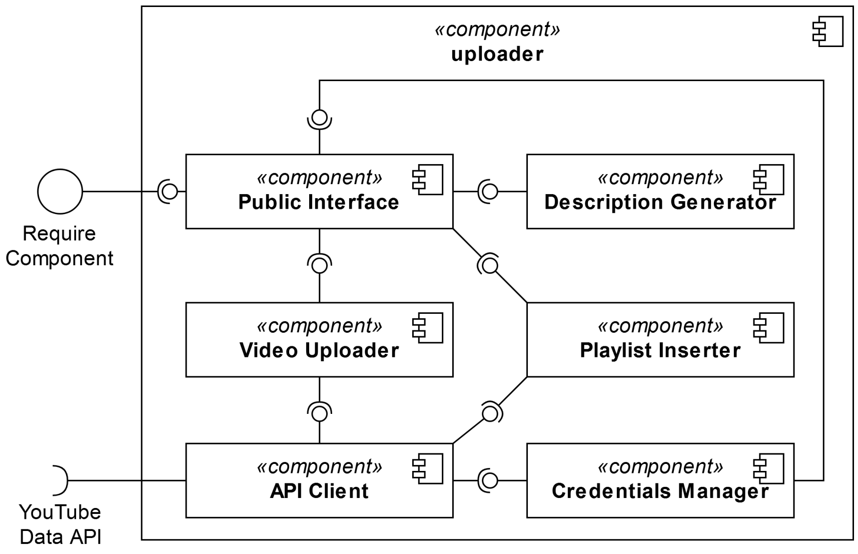


Figure 5.13: Uploader Module UML Component Diagram

5.4 Parallelization

The lecture.js pipeline performs all primary tasks in sequential order as most of these primary tasks require data generated by previously executed primary tasks. However, the pipeline also supports data-parallel parallelization within specific primary tasks by dividing them into sub-tasks. For example, the generation of frames is a primary task. In contrast, the generation of an individual frame would be a sub-task that could be executed parallel to a similar sub-task, like generating a different individual frame.

These sub-tasks operate on the same set of data but are independent of each other. Because of this, they can be parallelized in a limited manner so that each sub-task performs a similar but distinct operation using its copy or section of the complete data.

For this purpose, a request manager was created as a global script accessible by all pipeline modules. This request manager manages the execution of general tasks as requests in a data-parallel manner.

Each sub-task is defined in a request object that consists of a synchronous or asynchronous `function` to be called with the defined `parameters` by the request manager. Once the request is completed, a synchronous or asynchronous `callback` function is called with the output from the `function` and the defined `callback_parameters`.

```
const requests = [];
const request = {
    function : someFunction,
    parameters : [123, 'abc'],
    callback : (output, numbers, letters) => {
        console.log(output); // output from someFunction()
        console.log(numbers); // 456
        console.log(letters); // 'def'
    },
    callback_parameters : [456, 'def']
};
requests.push(request);
```

The request manager is then called with the array of request objects and a set of options. The option `max_concurrent` defines how many requests of the batch may be run at any one point in time and `max_per_second` how many requests may be started at most per second.

```
const requester = require('./requester.js');
await requester.run(
    requests,
    { max_concurrent : 3,
    max_per_second : 2 }
);
```

The request manager will then execute the requests in the order that they appear in the array asynchronously in parallel within the limitations defined in the options object. The request manager will return a promise that is only resolved once all requests were successfully executed. The `await` keyword in the code example above indicates that the program will wait for the promise to be resolved, which means that the pipeline will only continue running once all sub-tasks are completed. Within the pipeline, the following tasks were parallelized:

- **Video resources**: conversion to the right codec; trimming; manipulation of volume and speed
- **Audio resources**: trimming; manipulation of volume and speed
- **Text-to-Speech**: requests to the APIs
- **Frame generation**: extraction from PDFs or videos; copying of external image resources
- **Video generation**: combining frames and audio into individual clips

As an example of parallelization in the pipeline, the diagram in figure 5.14 shows the data-parallelized sub-tasks of converting embedded external video resources.

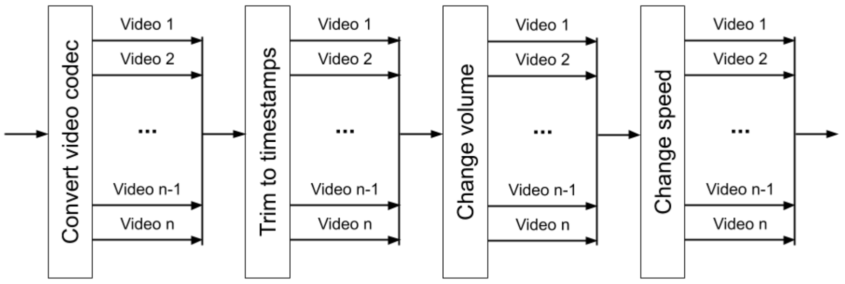


Figure 5.14: Data-parallel Video Resource Conversion

It would have also been possible to parallelize specific primary tasks that are entirely independent of each other in a task-parallel manner. For example, the extraction of image frames and the generation of Text-to-Speech audio files could run at the same time because they do not require any data generated by each other, as shown in figure 5.15. However, this would have impeded the performance of the sub-task parallelization within these primary tasks, making the overall performance gain negligible while sharply increasing the software's complexity. Instead, running all primary tasks in a sequential manner ensures that the resulting output to the user's terminal follows a logical order. This improves the user's overview of the program flow.

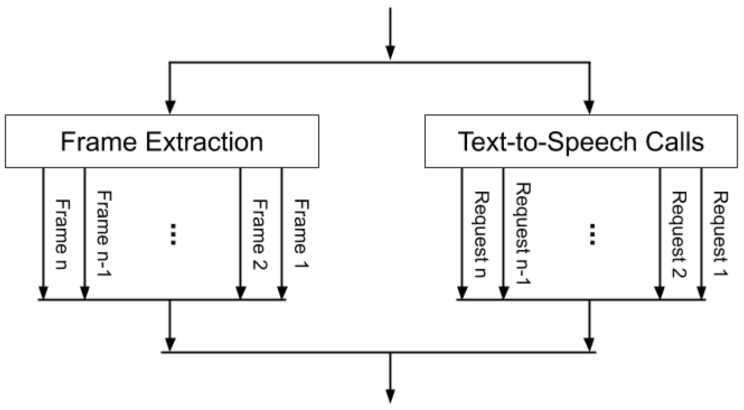


Figure 5.15: Potential Task-parallel Frame Extraction and Text-to-Speech

5.5 Logging

All modules use a custom logging script that extends JavaScript's built-in debugging console. The custom logger provides several semantic log types that may be used depending on the context of a log. The following log types exist:

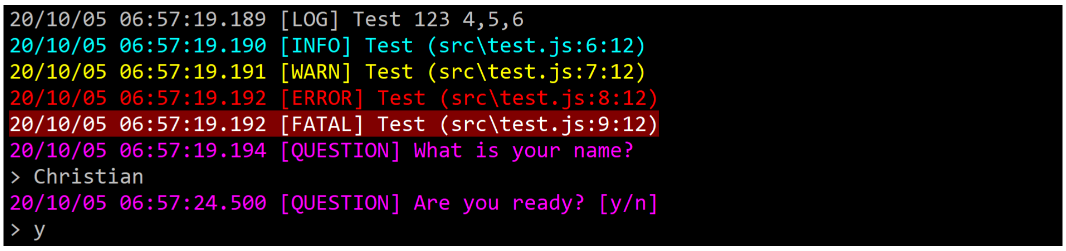
- `message` : general message
- `info` : informative message
- `warn` : warning message
- `error` : error message
- `fatal` : fatal error message, that should be followed by the termination of the program
- `question` : question that the user may answer using the terminal
- `confirm` : question to which the user must respond `y` (yes) or `n` (no)

The different types of logs are color-coded and formatted using ANSI escape codes that work in both Linux and Windows terminals. When a log is printed, it displays the exact time when it was invoked. Warnings, errors, and informative messages also provide the script and line number where they were invoked.

The idea behind the custom logger script is to add as much information as possible to each log while keeping them short and clean. In addition, the logger script also saves each invoked log as an object and provides several functions for querying and filtering the saved logs. It also supports the creation of log files. This function is used by the pipeline to output a log file into the output folder after each execution of the pipeline.

The following code example would produce the terminal output shown in figure 5.16.

```
var logger = require('./logger.js');
(async function() {
    logger.message('Test', 123, [4, 5, '6']);
    logger.info('Test');
    logger.warn('Test');
    logger.error('Test');
    logger.fatal('Test');
    await logger.question('What is your name?');
    await logger.confirm('Are you ready?');
})();
```



```
20/10/05 06:57:19.189 [LOG] Test 123 4,5,6
20/10/05 06:57:19.190 [INFO] Test (src\test.js:6:12)
20/10/05 06:57:19.191 [WARN] Test (src\test.js:7:12)
20/10/05 06:57:19.192 [ERROR] Test (src\test.js:8:12)
20/10/05 06:57:19.192 [FATAL] Test (src\test.js:9:12)
20/10/05 06:57:19.194 [QUESTION] What is your name?
> Christian
20/10/05 06:57:24.500 [QUESTION] Are you ready? [y/n]
> y
```

Figure 5.16: Logs displayed in the Windows 10 terminal

5.6 Style Guide

The source code must be consistent over the many modules and scripts that were implemented, which is why a basic style guide was specified during development. For context in the following sections, a script refers to a single JavaScript file. In contrast, a module refers to one or multiple scripts related to one another, which are typically imported as a single module.

Script Structure

In lecture.js, JavaScript files should use the *strict* mode, which restricts the language by using more strict semantics and disabling some of JavaScript's quirky features. The *strict* mode causes some otherwise silent errors to be thrown explicitly and prevents the use of some syntax that is likely to be defined in future versions of JavaScript. The *strict* mode can be enabled by placing the following string at the top of the file:

```
'use strict';
```

Then, any modules imported into the script are defined. The module definitions should follow a specific order, where first come the internal Node modules, then global scripts, followed by other scripts from the same module. Lastly, other modules and external libraries may be imported. To not pollute the file's global namespace with variables, an object is defined that references all the modules, which is then used to call the modules. The object has a variable name of one character, which is `_`.

```
const _ = {};
_.fs = require('fs');
_.logger = require('./logger.js');
_.logger.message('This is a log.');
```

Another part of a script's header is the definition of variables with script-wide scope. These variables may be used from any place within that particular script. Any constants with file-wide scope must be defined with capital letters, followed by other non-constant variable definitions with file-wide scope. Whenever possible, variables that will not change in type should be defined as constants with the keyword `const`, while other variables definitions should use the keyword `let`. Each variable definition should use a separate variable keyword instead of defining multiple variables with a single keyword. The variable keyword `var` should be avoided because it has special scoping rules, where it is scoped to the beginning of the immediate enclosing function. This means that if the variable is defined at the end of the file using `var`, it is also accessible at the beginning.

```
// constants
const EOL = _.os.EOL;
// definitions
let COUNTER = 0;
```

Next come the definitions of private functions, which are only available from within the script. After these, the definition of an object with the name `__public` follows. This public object holds all public functions, each of which gets a key assigned by which it may be referenced from outside of the object. At the end of the script, the public object is exported and becomes available to other scripts that import the script.

```
// private functions
const privateFunc = () => { ... };
// public functions
const __public = {
    publicFunc : () => { ... }
};
module.exports = __public;
```

Function Structure

There should be a documentation comment defined above every function using the JSDoc syntax. This is explained in more detail in section 5.7.

Any function parameters should be tested for the correct type and valid content when possible. These type tests should be separated into sections for required and optional function parameters, although both sections must be located at the top of the function body before any other content.

Functions should return the expected value from a successful execution using the last return statement at the bottom of the function body, if possible. Return statements triggered by an error should appear above the successful return statements inside the function body.

Syntax

Local variables inside functions and object keys should use the snake case syntax, while configuration values and function names should use the title case syntax.

Longer strings should be generally defined using backticks, which allows the insertion of variables directly into the string without concatenation. Strings that are a few characters long may also be defined using apostrophes. Quotation marks are not to be used to define strings.

Wherever possible, the simple syntax of ECMAScript 6 should be used, which includes the definitions of functions. Because Node.js supports the syntax and runs outside the web browser, backward compatibility with older web browsers, which often prevents the usage of ECMAScript 6 on modern websites, is not an issue for this desktop software implementation.

```
// ECMAScript 5
function name(variable) { ... }
// ECMAScript 6
const name = variable => { ... }
```

Most asynchronous Node.js functions implement a function parameter, which should be a callback function to be executed once the asynchronous function finished its execution. These callback functions are called with an error parameter, which includes information about possible errors from when the asynchronous function was executed. They also have a second parameter that transfers the output from the asynchronous function. This callback structure of asynchronous functions in Node.js should be avoided in lecture.js. Instead, JavaScript Promises, `then` functions and the keywords `async` and `await` should be used.

Whenever applicable, the synchronous variant of an asynchronous Node.js function should be used. If that is not possible, the asynchronous variant should be wrapped into a promise whose fulfillment is to be synchronously awaited using the `await` keyword.

5.7 Documentation

All modules and functions in the source code were annotated using documentation comments with the syntax of JSDoc. JSDoc[13] is a documentation generator for JavaScript that enables the automatic generation of an API documentation from annotated source code.

Besides the advantage of having an automatic API documentation, the syntax of the documentation comments is also very similar to JavaDoc[14]. JavaDoc is a documentation generator for Java utilized in a wide range of software projects and of which variants exist for a multitude of programming languages. This means that the documentation syntax is known to a wide range of programmers, besides being easy to read in general. An example of a JSDoc documentation comment used in the software looks like the following:

```
/**
* checks if a voice exists
*
* @async
* @function
* @alias module:text_to_speech
* @category public
*
* @param {string} voice_name - name of the voice
* @returns {Promise.<boolean>} true if the voice exists
*/
voiceExists : async(voice_name) => {
    ...
}
```

A documentation module was created that scans the source directory for all JavaScript files. With the help of the jsdoc2md[15] library, it compiles the results into a markdown documentation file. Documentation generators like JSDoc often focus on generating Hypertext Markup Language (HTML) files. However, in contrast to HTML, markdown files make it possible to directly view the documentation in the repository on GitLab or GitHub, without the need to host an additional website to render HTML content.

[13] JSDoc: jsdoc.app
[14] JavaDoc: docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html
[15] jsdoc2md: github.com/jsdoc2md/jsdoc-to-markdown

Chapter 6

Evaluation

Multiple groups were invited to prepare a pre-recorded presentation for the PAN workshop of the CLEF 2020 conference. These groups and some additional individuals were contacted to participate in a user study evaluating lecture.js by producing a video presentation[1] or trying out the tool privately. Four participants provided feedback in an online questionnaire. It included 14 questions covering ease of usage of lecture.js, perceived time savings, quality of the Text-to-Speech output, and perceived limitations with suggestions for improvements.[2] The study participants had access to a GitHub repository with the source code and credentials files for Amazon Polly and Google Cloud Text-to-Speech. An installation guide, a *How to use* guide, and documentation of all markup features were provided. Although technical support was offered, three out of four participants completed the study without additional help. Although the study provided some great first insights into the feasibility and possible problems of lecture.js, it is not completely representative of the target user groups of lecture.js because there were only four participants.

Participants familiarized themselves with the software fairly quickly. The installation process took less than half an hour for most participants (figure 6.1). Familiarization with the software's basic usage took one to two hours on average (figure 6.2).

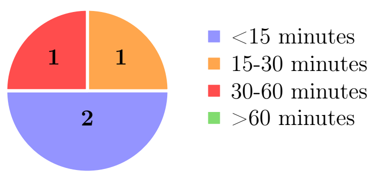


Figure 6.1: Installation time

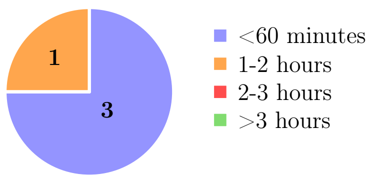


Figure 6.2: Familiarization time

[1] Example video generated in the user study: youtu.be/699e_l6XDI8
[2] The questions and results are listed in the appendix starting on page I.

However, the participants felt that only a little time was saved using the software compared to recording and editing a video themselves (figure 6.3). Although there are only some perceived time savings, this may improve once users have more experience using the software or access to a graphical user interface. The participants judged the learning of the software's basics as generally easy (figure 6.4). Simultaneously, they described the difficulty of using the software to generate videos as moderate, though with a slight tendency to easy (figure 6.5). However, it must be taken into account that all participants are very experienced in computer science, which will not apply to the software's whole target user group.

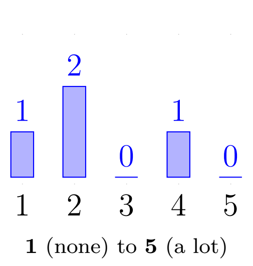


Figure 6.3: Perceived time savings

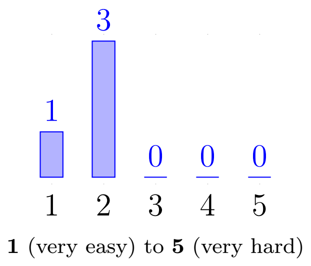


Figure 6.4: Difficulty understanding the basics

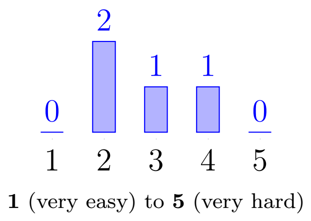


Figure 6.5: Difficulty using the software

The participants were, in general, satisfied with the quality of the Text-to-Speech output, describing the quality of voices as moderate (figure 6.6) and the number of available languages as enough (figure 6.7).

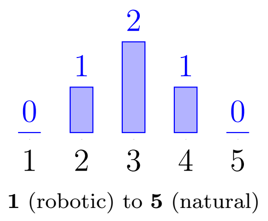


Figure 6.6: Voice sound quality

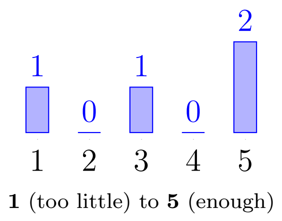


Figure 6.7: Available languages

There was no clear preference for voices from Google Cloud Text-to-Speech or Amazon Polly (figure 6.8). This is surprising because, on the whole, Amazon Polly's voices sound audibly worse than Google Cloud's voices. However, this might be explained by the participants only using Amazon Polly's new neural voices, which can approach Google Cloud's WaveNet's voices in terms of quality.

Additionally, participants had to correct the pronunciation of very few words in their video presentations (figure 6.9).

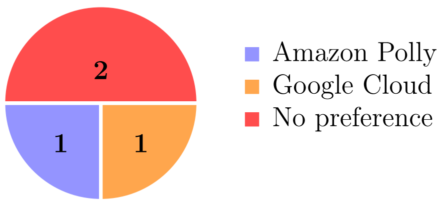


Figure 6.8: API preferences

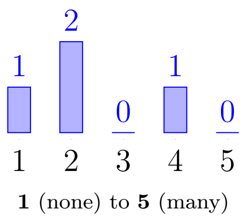


Figure 6.9: Word mispronunciations

The participants felt somewhat limited in what they could achieve when using the software (figure 6.10). There were several suggestions for improvements and additional features. One participant suggested specific simplifications of the markup language, which were implemented by removing some unnecessary elements, making some required elements and attributes entirely optional, and re-naming some elements and attributes. Additionally, an LSML markup element for inserting images and an attribute for fitting media elements to the output resolution was added after a participant suggestion. Some other suggestions were the following, which could be implemented in the future:

- overlaying an audio file over an inserted video
- preview of individual sections of the lecture
- parsing named destinations in PDF documents, where the pages with destinations can later be loaded as slides using the corresponding anchor tags

In total, the participants held a positive attitude towards the software. They indicated it would be likely for them to use the software again to generate video presentations in the future (figure 6.11).

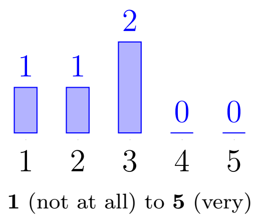


Figure 6.10: Perceived limitation in the usage

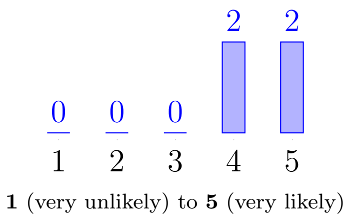


Figure 6.11: Likelihood to use lecture.js again

Chapter 7

Conclusion and Future Work

Due to the COVID-19 pandemic and its impact on people worldwide, pre-recorded lectures recently experienced a rise in popularity. However, creating high-quality video lectures can be a difficult, time-consuming, and costly task. That is why this thesis aimed to develop a tool that would enable teachers to generate spoken video lectures from their scripts and slides while removing the need for manual audio recording and video editing. For this purpose, the custom lecture-markup language LSML was proposed and implemented as an extension of SSML with many features for Text-to-Video generation. In addition to language and prosodic features of SSML, LSML supports the integration of slides, external visual resources, and video manipulation. LSML scripts are validated, pre-processed, and parsed by a custom software implementation called lecture.js created in the course of this thesis. The resulting data is used to request Amazon Polly and Google Cloud Text-to-Speech and generate natural-sounding speech. The Text-to-Speech audio is then combined with PDF slides and external video, audio, and image resources provided by the user to generate a video lecture, which can be directly uploaded to YouTube.

A user study was conducted after the mandatory features were functional to evaluate the user experience that lecture.js provides. The participants rated the software to be fairly straightforward to use. They achieved some time savings, which can undoubtedly be improved upon with future quality of life features and if the users have more experience working with the program. Most participants did not dislike the Text-to-Speech output, and all indicated it to be likely for them to use the software to generate more video presentations in the future. In the end, lecture.js succeeded in its initial goal to enable its users to save both time and money when creating video lectures. However, even more requirements for lecture.js were outlined in this thesis's planning stages. All those requirements deemed to be primary were successfully met, and all mandatory features were implemented.

Some of the supplemental features could not be implemented in time. This leaves room for several aspects of the software that may be extended or improved in the future. The most pressing issue is the missing graphical user interface, which, once implemented, will lower the learning curve of the software even further and enable teachers to create video lectures more efficiently. Additionally, the highlighting of sections in the slides for improved interactivity would only be efficiently useable when combined with a graphical user interface. Another important supplemental feature that will need to be added is the support for subtitles, which will enable deaf or hard-of-hearing students to follow the lecture more easily.

Additional Text-to-Speech services can also be integrated into the software to increase the pool of available voices and languages. One of the examples for such a service is the open-source software OpenMARY. Although OpenMARY's voices sound moderately robotic, the project should still be integrated into lecture.js in the future because it has two significant advantages. OpenMARY is free, and it supports most SSML markup features. This makes the software ideal for testing lecture.js and previewing generated lectures without accruing unnecessary costs from the commercial Text-to-Speech services.

However, even without these features, lecture.js is already usable by teachers in real-life environments, as shown by the user study. A further significant advantage of lecture.js is that it requires very little maintenance unless there are substantial changes in the employed APIs and libraries. Although the quality of the voices was rated as moderate by the user study participants, the software will continually improve its audio output due to Amazon Polly and Google Cloud Text-to-Speech improving theirs. This gives the quality of the audio output great future potential with little to no additional work.

In the present, computer-generated voices still sound distinctly fake. They are not as robotic as they used to be, though, and some are already successfully employed and well-received in real-world applications. The field of Text-to-Speech generation is making rapid progress in improving the sound quality and life-like attributes of computer-generated voices. The last few years saw some especially promising developments in terms of generating natural-sounding voices using machine learning. This development will undoubtedly continue in the future and possibly elevate computer-generated voices to a level where they are virtually indistinguishable from human speech. Perhaps within the next decade, computer-generated content will be accepted in education environments just as much as their human counterpart. These technological advancements spell great news for software like lecture.js, which can leverage this technology most effectively.

Bibliography

Alexa Siterank Competitive Analysis. YouTube Siterank Analysis. https://www.alexa.com/siteinfo/youtube.com, 2020. accessed on 28.09.2020. 1

AWS Machine Learning Blog. Amazon Polly Neural Text-to-Speech voices now available in Sydney Region. https://aws.amazon.com/blogs/machine-learning/amazon-polly-neural-text-to-speech-voices-now-available-in-sydney-region/, 2019. accessed on 04.09.2020. 5.3.5

J. Couperthwaite, W. E. Leadbeater, and K. Nightingale. Evaluating the use and impact of lecture recording in undergraduates: Evidence for distinct approaches by different groups of students. https://www.researchgate.net/publication/233406488_Evaluating_the_use_and_impact_of_lecture_recording_in_undergraduates_Evidence_for_distinct_approaches_by_different_groups_of_students, 2012. 1, 2

A. J. Danielson, V. Preast, H. Bender, and L. Hassall. Is the effectiveness of lecture capture related to teaching approach or content type? https://www.researchgate.net/publication/259127182_Is_the_effectiveness_of_lecture_capture_related_to_teaching_approach_or_content_type, 2014. 2

R. M. Hadgu, S. H. Huynh, and C. Gopalan. The Use of Pre-recorded Lectures on Student Performance in Physiology. https://files.eric.ed.gov/fulltext/EJ1157564.pdf, 2016. 1, 2

Ministry for Culture, Youth and Sport of Baden Württemberg. Video conference tool for schools in the south west. https://km-bw.de/,Lde/Startseite/Service/2020+06+22+Big+Blue+Button+und+Fortbildungsangebote, 2020. accessed on 28.09.2020. 1

E. Nordmann and P. McGeorge. Lecture capture in higher education: Time to learn from the learners.

https://www.researchgate.net/publication/324943672_Lecture_capture_in_higher_education_time_to_learn_from_the_learners, 2018. 1

A. J. Prunuske, J. Batzli, Howell E., and S. Miller. Using Online Lectures to Make Time for Active Learning. https://www.genetics.org/content/192/1/67, 2012. 2

The Guardian. Zoom booms as demand for video-conferencing tech grows. https://www.theguardian.com/technology/2020/mar/31/zoom-booms-as-demand-for-video-conferencing-tech-grows-in-coronavirus-outbreak, 2020. accessed on 28.05.2020. 1

UNESCO. COVID-19 Educational Disruption and Response. https://en.unesco.org/covid19/educationresponse, 2020. accessed on 28.05.2020. 1

World Wide Web Consortium. Cascading Style Sheets Level 2 Recommendation. https://www.w3.org/TR/1998/REC-CSS2-19980512/, 1998. accessed on 31.08.2020. 4.6

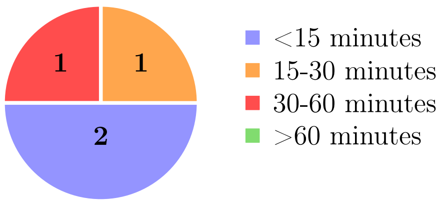
World Wide Web Consortium. SSML Specification v1.1. https://www.w3.org/TR/speech-synthesis11/, 2010. accessed on 31.08.2020. 4, 4.3, 5.3.2

Appendices

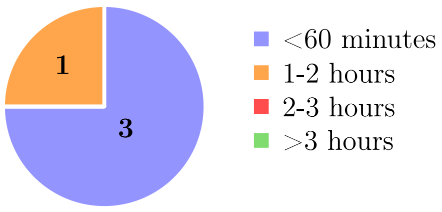
User Study Questionnaire

The questionnaire was provided as a Google Form after the user study with the following questions and results. There were 4 participants in total.

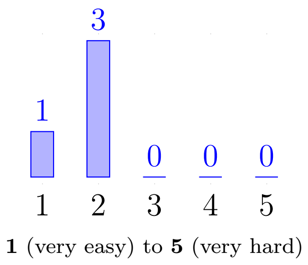
1. **How long did it take you to install the software?**



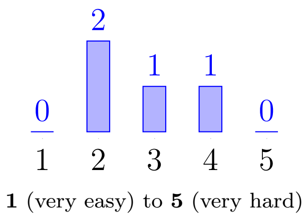
2. **How long did it take you to familiarize yourself with the software?**



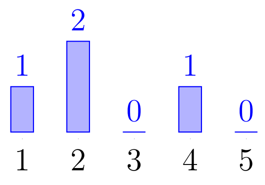
3. **How difficult was learning the basics of the software?**



4. **How difficult would you describe using the software?**

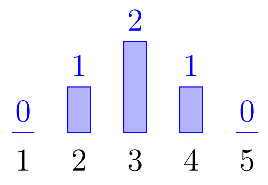


5. **Do you feel you saved time using the software compared to recording and editing a video yourself?**



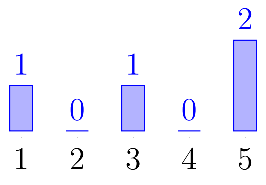
1 (none) to **5** (a lot)

6. **How would you describe the quality of the voices?**



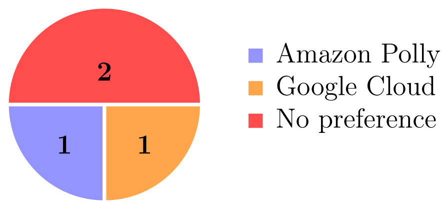
1 (robotic) to **5** (natural sounding)

7. **Is the amount of available languages enough?**

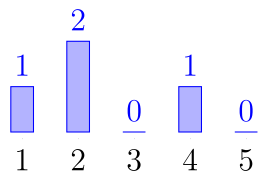


1 (too little) to **5** (more than enough)

8. **Did you prefer Amazon's or Google's voices?**

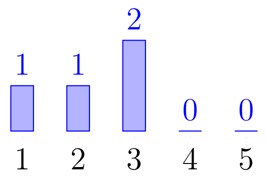


9. **Did you have to correct the pronunciation of words?**



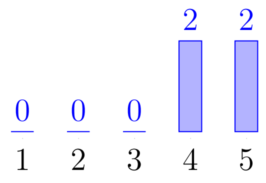
1 (none at all) to **5** (many)

10. **Did you feel limited in how you could use the software?**



1 (not at all) to **5** (very)

11. **How likely is it that you would use the software again for generating video presentations?**



1 (very unlikely) to **5** (very likely)

12. **Was there something that you wanted to do for the video, which was not possible with the software?**

13. **Did you happen upon any problems using the software?**

14. **Do you have any suggestions for improvement?**

Simple Example Script

```
<lecture>

<info title="Simple Example" />

<settings voice="amazon-en-gb-amy" />

<deck id="slides1" src="slides1.pdf" active="true" />
<deck id="slides2" src="slides2.pdf" />

Welcome to a short example that will demonstrate the basic
   features of the software.

Currently, slide 1 is set as the active slide deck.

<slide page="+1" />

Now we are on the second page of slide 1.
We can also change the slide deck that is open!

<slide deck="slides2" page="1" />

It is also possible to change languages. For a test, this will
   be <de>ausgesprochen in Deutsch</de>. It spoke German with
   an English accent.

We can also add breaks and a lot of other cool stuff!

<break time="1500ms" />

It is possible to insert images!

<image src="test1.png" />

This program can also play audio files.

<audio src="audio-example.m4a" />

And videos too!

<video src="video-example.mp4" />

</lecture>
```

LSML XSD Schema

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:ssml="http://www.w3.org/2001/10/synthesis"
           xmlns="https://example.com/lecture.js"
           targetNamespace="https://example.com/lecture.js"
           elementFormDefault="qualified">

    <xs:annotation>
        <xs:documentation>
            LSML Schema
        </xs:documentation>
    </xs:annotation>

    <!-- import SSML schema https://www.w3.org/TR/speech-synthesis11/#AppD -->
    <xs:import namespace="http://www.w3.org/2001/10/synthesis" schemaLocation="synthesis.xsd" />

    <!-- import dependent namespaces, e.g., xml:id -->
    <xs:import namespace="http://www.w3.org/XML/1998/namespace" schemaLocation="xml.xsd"/>

    <!-- LSML types -->

    <xs:simpleType name="positiveInteger">
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="1" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="atLeastOneCharacter">
        <xs:restriction base="xs:string">
            <xs:pattern value=".+"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="absoluteOrRelativePath">
        <xs:annotation>
            <xs:documentation>
                ignore Linux absolute paths starting with a,
                tilde (~) because lecture.js can not handle them
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="[^~]{1}.*"/>
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="timestamp">
        <xs:annotation>
            <xs:documentation>
                defines a timestamp
                    1) FFmpeg format: 23:59:59 or 23:59:59.999
                    2) SSML format: 5s or 5000ms
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="[0-9]{2}\:[0-9]{2}\:[0-9]{2}(\.[0-9]{3})?|[0-9]+s|[0-9]+ms" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="googleEffectProfile">
        <xs:annotation>
            <xs:documentation>
                defines valid effect profiles for Google Cloud Text-to-Speech:
                https://cloud.google.com/text-to-speech/docs/audio-profiles#available_audio_profiles
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:enumeration value="wearable-class-device" />
            <xs:enumeration value="handset-class-device" />
            <xs:enumeration value="headphone-class-device" />
            <xs:enumeration value="small-bluetooth-speaker-class-device" />
            <xs:enumeration value="medium-bluetooth-speaker-class-device" />
            <xs:enumeration value="large-home-entertainment-class-device" />
            <xs:enumeration value="large-automotive-class-device" />
            <xs:enumeration value="telephony-class-application" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="youtubePrivacyStatus">
        <xs:annotation>
            <xs:documentation>
                defines a privacy status for a video when uploaded to YouTube
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:enumeration value="public" />
            <xs:enumeration value="unlisted" />
            <xs:enumeration value="private" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="resourceSpeed">
        <xs:annotation>
            <xs:documentation>
                attribute that defines the speed of an embedded audio
                file as a percentage value between 50% and 200%
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="([5-9][0-9]|1[0-9][0-9]|200)%" />
        </xs:restriction>
    </xs:simpleType>

    <xs:simpleType name="resourceSoundLevel">
        <xs:annotation>
            <xs:documentation>
                attribute that defines the volume of an embedded audio file
            </xs:documentation>
        </xs:annotation>
        <xs:restriction base="xs:string">
            <xs:pattern value="(\+|\-)(([1-4]?[0-9]{1})|50)dB" />
        </xs:restriction>
    </xs:simpleType>

    <!-- The following LSML language elements are dynamically determined
        at run-time using the available languages for the voice APIs.
        They may need to be updated from time to time!
        This includes the main definitions here, as well as
        the references to them further down below! -->

    <xs:element name="ar"/><xs:element name="ar-XA"/><xs:element name="arb"/><xs:element name="bn"/><
     xs:element name="bn-IN"/><xs:element name="cmn"/><xs:element name="cmn-CN"/><xs:element name="cmn-
     TW"/><xs:element name="cs"/><xs:element name="cs-CZ"/><xs:element name="cy"/><xs:element name="cy-
     GB"/><xs:element name="da"/><xs:element name="da-DK"/><xs:element name="de"/><xs:element name="de-
     DE"/><xs:element name="el"/><xs:element name="el-GR"/><xs:element name="en"/><xs:element name="en-
     AU"/><xs:element name="en-GB"/><xs:element name="en-GB-WLS"/><xs:element name="en-IN"/><xs:element
      name="en-US"/><xs:element name="es"/><xs:element name="es-ES"/><xs:element name="es-MX"/><
     xs:element name="es-US"/><xs:element name="fi"/><xs:element name="fi-FI"/><xs:element name="fil"/>
     <xs:element name="fil-PH"/><xs:element name="fr"/><xs:element name="fr-CA"/><xs:element name="fr-
     FR"/><xs:element name="gu"/><xs:element name="gu-IN"/><xs:element name="hi"/><xs:element name="hi-
     IN"/><xs:element name="hu"/><xs:element name="hu-HU"/><xs:element name="id"/><xs:element name="id-
     ID"/><xs:element name="is"/><xs:element name="is-IS"/><xs:element name="it"/><xs:element name="it-
     IT"/><xs:element name="ja"/><xs:element name="ja-JP"/><xs:element name="kn"/><xs:element name="kn-
     IN"/><xs:element name="ko"/><xs:element name="ko-KR"/><xs:element name="ml"/><xs:element name="ml-
     IN"/><xs:element name="nb"/><xs:element name="nb-NO"/><xs:element name="nl"/><xs:element name="nl-
     NL"/><xs:element name="pl"/><xs:element name="pl-PL"/><xs:element name="pt"/><xs:element name="pt-
     BR"/><xs:element name="pt-PT"/><xs:element name="ro"/><xs:element name="ro-RO"/><xs:element name="
     ru"/><xs:element name="ru-RU"/><xs:element name="sk"/><xs:element name="sk-SK"/><xs:element name="
     sv"/><xs:element name="sv-SE"/><xs:element name="ta"/><xs:element name="ta-IN"/><xs:element name="
     te"/><xs:element name="te-IN"/><xs:element name="th"/><xs:element name="th-TH"/><xs:element name="
     tr"/><xs:element name="tr-TR"/><xs:element name="uk"/><xs:element name="uk-UA"/><xs:element name="
     vi"/><xs:element name="vi-VN"/><xs:element name="yue"/><xs:element name="yue-HK"/>

    <!-- LSML custom elements -->

    <xs:element name="slide">
        <xs:annotation>
            <xs:documentation>
                switches to another slide deck and/or page
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:attribute name="page" use="required">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:pattern value="(\+|\-)?[0-9]+|next|previous|first|last" />
                    </xs:restriction>
                </xs:simpleType>
            </xs:attribute>
            <xs:attribute name="deck" type="xs:string" use="optional" />
            <xs:attribute name="fit" use="optional">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="contain" />
                        <xs:enumeration value="cover" />
                        <xs:enumeration value="fill" />
                    </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:element>

<xs:element name="video">
    <xs:annotation>
        <xs:documentation>
            inserts a video resource
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:attribute name="src" type="absoluteOrRelativePath" use="required" />
        <xs:attribute name="keepFrame" type="xs:boolean" use="optional" />
        <xs:attribute name="clipBegin" type="timestamp" use="optional" />
        <xs:attribute name="clipEnd" type="timestamp" use="optional" />
        <xs:attribute name="speed" type="resourceSpeed" use="optional" />
        <xs:attribute name="soundLevel" type="resourceSoundLevel" use="optional" />
        <xs:attribute name="repeatCount" type="positiveInteger" use="optional" />
        <xs:attribute name="fit" use="optional">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="contain" />
                    <xs:enumeration value="cover" />
                    <xs:enumeration value="fill" />
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:element>

<xs:element name="image">
    <xs:annotation>
        <xs:documentation>
            inserts an image resource
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:attribute name="src" type="absoluteOrRelativePath" use="required" />
        <xs:attribute name="fit" use="optional">
            <xs:simpleType>
                <xs:restriction base="xs:string">
                    <xs:enumeration value="contain" />
                    <xs:enumeration value="cover" />
                    <xs:enumeration value="fill" />
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:element>

<!-- modified SSML elements -->

<xs:element name="lexicon">
    <xs:annotation>
        <xs:documentation>
            defines a lexicon and assigns it an ID
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="lexeme" minOccurs="1" maxOccurs="unbounded">
                <xs:annotation>
                    <xs:documentation>
                        defines a lexeme (basic lexical unit) inside a lexicon
                    </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                    <xs:choice minOccurs="2" maxOccurs="2">
                        <xs:element name="grapheme" type="xs:string" minOccurs="1" maxOccurs="1" />
                        <xs:element name="alias" type="xs:string" maxOccurs="1" />
                        <xs:element name="phoneme" type="xs:string" maxOccurs="1" />
                    </xs:choice>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute ref="xml:id" use="required" />
        <xs:attribute name="alphabet" type="xs:string" use="optional" />
    </xs:complexType>
</xs:element>

<xs:element name="voice">
    <xs:annotation>
        <xs:documentation>
```

```
                changes the narrating voice
            </xs:documentation>
        </xs:annotation>
        <xs:complexType mixed="true">
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="slide" />
                <xs:element ref="video" />
                <xs:element ref="image" />
                <xs:element ref="audio" />
                <xs:element ref="mark" />
                <xs:element ref="say-as" />
                <xs:element name="p" type="ssml:paragraph" />
                <xs:element name="s" type="ssml:sentence" />
                <xs:element name="token" type="ssml:tokenType" />
                <xs:element name="w" type="ssml:tokenType" />
                <xs:element name="lang" type="ssml:langType" />
                <xs:element name="prosody" type="ssml:prosody" />
                <xs:element name="emphasis" type="ssml:emphasis" />
                <xs:element name="sub" type="ssml:sub" />
                <xs:element name="phoneme" type="ssml:phoneme" />
                <xs:element name="break" type="ssml:break" />
                <xs:element name="lookup" type="ssml:lookupType" />

                <!-- LSML Language Codes (see explanation above) -->

                <xs:element ref="ar"/><xs:element ref="ar-XA"/><xs:element ref="arb"/><xs:element ref="
     bn"/><xs:element ref="bn-IN"/><xs:element ref="cmn"/><xs:element ref="cmn-CN"/><xs:element ref="
     cmn-TW"/><xs:element ref="cs"/><xs:element ref="cs-CZ"/><xs:element ref="cy"/><xs:element ref="cy-
     GB"/><xs:element ref="da"/><xs:element ref="da-DK"/><xs:element ref="de"/><xs:element ref="de-DE"/
     ><xs:element ref="el"/><xs:element ref="el-GR"/><xs:element ref="en"/><xs:element ref="en-AU"/><
     xs:element ref="en-GB"/><xs:element ref="en-GB-WLS"/><xs:element ref="en-IN"/><xs:element ref="en-
     US"/><xs:element ref="es"/><xs:element ref="es-ES"/><xs:element ref="es-MX"/><xs:element ref="es-
     US"/><xs:element ref="fi"/><xs:element ref="fi-FI"/><xs:element ref="fil"/><xs:element ref="fil-PH
     "/><xs:element ref="fr"/><xs:element ref="fr-CA"/><xs:element ref="fr-FR"/><xs:element ref="gu"/><
     xs:element ref="gu-IN"/><xs:element ref="hi"/><xs:element ref="hi-IN"/><xs:element ref="hu"/><
     xs:element ref="hu-HU"/><xs:element ref="id"/><xs:element ref="id-ID"/><xs:element ref="is"/><
     xs:element ref="is-IS"/><xs:element ref="it"/><xs:element ref="it-IT"/><xs:element ref="ja"/><
     xs:element ref="ja-JP"/><xs:element ref="kn"/><xs:element ref="kn-IN"/><xs:element ref="ko"/><
     xs:element ref="ko-KR"/><xs:element ref="ml"/><xs:element ref="ml-IN"/><xs:element ref="nb"/><
     xs:element ref="nb-NO"/><xs:element ref="nl"/><xs:element ref="nl-NL"/><xs:element ref="pl"/><
     xs:element ref="pl-PL"/><xs:element ref="pt"/><xs:element ref="pt-BR"/><xs:element ref="pt-PT"/><
     xs:element ref="ro"/><xs:element ref="ro-RO"/><xs:element ref="ru"/><xs:element ref="ru-RU"/><
     xs:element ref="sk"/><xs:element ref="sk-SK"/><xs:element ref="sv"/><xs:element ref="sv-SE"/><
     xs:element ref="ta"/><xs:element ref="ta-IN"/><xs:element ref="te"/><xs:element ref="te-IN"/><
     xs:element ref="th"/><xs:element ref="th-TH"/><xs:element ref="tr"/><xs:element ref="tr-TR"/><
     xs:element ref="uk"/><xs:element ref="uk-UA"/><xs:element ref="vi"/><xs:element ref="vi-VN"/><
     xs:element ref="yue"/><xs:element ref="yue-HK"/>

            </xs:choice>
            <xs:attribute name="name" type="xs:string" use="required" />
        </xs:complexType>
    </xs:element>

    <xs:element name="say-as">
        <xs:annotation>
            <xs:documentation>
                interprets and says its contents in a certain way
            </xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:simpleContent>
                <xs:extension base="atLeastOneCharacter">
                    <xs:attribute name="interpret-as" use="required">
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="cardinal" />
                                <xs:enumeration value="ordinal" />
                                <xs:enumeration value="characters" />
                                <xs:enumeration value="spell-out" />
                                <xs:enumeration value="fraction" />
                                <xs:enumeration value="expletive" />
                                <xs:enumeration value="unit" />
                                <xs:enumeration value="date" />
                                <xs:enumeration value="time" />
                                <xs:enumeration value="telephone" />
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="format" type="xs:string" use="optional" />
                    <xs:attribute name="detail" type="xs:string" use="optional" />
                </xs:extension>
            </xs:simpleContent>
        </xs:complexType>
    </xs:element>
```

```
<xs:element name="audio">
    <xs:annotation>
        <xs:documentation>
            inserts an audio resource
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:attribute name="src" type="absoluteOrRelativePath" use="required" />
        <xs:attribute name="clipBegin" type="timestamp" use="optional" />
        <xs:attribute name="clipEnd" type="timestamp" use="optional" />
        <xs:attribute name="speed" type="resourceSpeed" use="optional" />
        <xs:attribute name="soundLevel" type="resourceSoundLevel" use="optional" />
        <xs:attribute name="repeatCount" type="positiveInteger" use="optional" />
    </xs:complexType>
</xs:element>

<xs:element name="mark">
    <xs:annotation>
        <xs:documentation>
            defines a marker
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:attribute name="name" type="xs:string" use="required" />
        <xs:attribute name="chapter" type="xs:string" use="optional" />
    </xs:complexType>
</xs:element>

<!-- LSML Root & Document Structure -->

<xs:element name="lecture">
    <xs:complexType mixed="true">
        <xs:choice maxOccurs="unbounded">

            <xs:element name="info" minOccurs="0" maxOccurs="1">
                <xs:annotation>
                    <xs:documentation>
                        defines information about the document
                    </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                    <xs:attribute name="title" type="xs:string" use="optional" />
                    <xs:attribute name="description" type="xs:string" use="optional" />
                    <xs:attribute name="authors" type="xs:string" use="optional" />
                    <xs:attribute name="copyright" type="xs:string" use="optional" />
                </xs:complexType>
            </xs:element>

            <xs:element name="settings" minOccurs="0" maxOccurs="1">
                <xs:annotation>
                    <xs:documentation>
                        defines settings for the generated video lecture
                    </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                    <xs:attribute name="voice" type="xs:string" use="optional" />
                    <xs:attribute name="resolution" use="optional">
                        <xs:simpleType>
                            <xs:annotation>
                                <xs:documentation>
                                    defines a video resolution in the format "{width}x{height}", e.g
 ., "1280x720"
                                </xs:documentation>
                            </xs:annotation>
                            <xs:restriction base="xs:string">
                                <!-- match width in range of 128 to 3840
                                     match height in range of 72 to 2160
                                    http://gamon.webfactional.com/regexnumericrangegenerator/
                                -->
                                <xs:pattern value="
 (12[89]|1[3-9][0-9]|[2-9][0-9]{2}|[12][0-9]{3}|3[0-7][0-9]{2}|38[0-3][0-9]|3840)x
 (7[2-9]|[89][0-9]|[1-8][0-9]{2}|9[0-8][0-9]|99[0-9]|1[0-9]{3}|20[0-9]{2}|21[0-5][0-9]|2160)" />
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="fps" use="optional">
                        <xs:simpleType>
                            <xs:restriction base="xs:integer">
                                <xs:minInclusive value="10" />
                                <xs:maxInclusive value="120" />
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="breakAfterSlide" type="positiveInteger" use="optional" />
                    <xs:attribute name="breakAfterParagraph" type="positiveInteger" use="optional" /
 >
                    <xs:attribute name="googleEffectProfile" type="googleEffectProfile" use="
 optional" />
                    <xs:attribute name="youtubePrivacyStatus" type="youtubePrivacyStatus" use="
 optional" />
                    <xs:attribute name="youtubePlaylistId" type="xs:string" use="optional" />
                </xs:complexType>
            </xs:element>

            <xs:element name="deck" minOccurs="1" maxOccurs="unbounded">
                <xs:annotation>
                    <xs:documentation>
                        initializes a PDF document with an ID
                    </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                    <xs:attribute name="id" type="xs:string" use="required" />
                    <xs:attribute name="src" type="absoluteOrRelativePath" use="required" />
                    <xs:attribute name="active" type="xs:boolean" use="optional" />
                    <xs:attribute name="fit" use="optional">
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="contain" />
                                <xs:enumeration value="cover" />
                                <xs:enumeration value="fill" />
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                </xs:complexType>
            </xs:element>

            <xs:element ref="lexicon" minOccurs="0" maxOccurs="unbounded" />

            <xs:element ref="slide" />
            <xs:element ref="video" />
            <xs:element ref="image" />
            <xs:element ref="audio" />
            <xs:element ref="voice" />
            <xs:element ref="mark" />
            <xs:element ref="say-as" />
            <xs:element name="p" type="ssml:paragraph" />
            <xs:element name="s" type="ssml:sentence" />
            <xs:element name="token" type="ssml:tokenType" />
            <xs:element name="w" type="ssml:tokenType" />
            <xs:element name="lang" type="ssml:langType" />
            <xs:element name="prosody" type="ssml:prosody" />
            <xs:element name="emphasis" type="ssml:emphasis" />
            <xs:element name="sub" type="ssml:sub" />
            <xs:element name="phoneme" type="ssml:phoneme" />
            <xs:element name="break" type="ssml:break" />
            <xs:element name="lookup" type="ssml:lookupType" />

            <!-- LSML Language Codes (see explanation above) -->

            <xs:element ref="ar"/><xs:element ref="ar-XA"/><xs:element ref="arb"/><xs:element ref="
 bn"/><xs:element ref="bn-IN"/><xs:element ref="cmn"/><xs:element ref="cmn-CN"/><xs:element ref="
 cmn-TW"/><xs:element ref="cs"/><xs:element ref="cs-CZ"/><xs:element ref="cy"/><xs:element ref="cy-
 GB"/><xs:element ref="da"/><xs:element ref="da-DK"/><xs:element ref="de"/><xs:element ref="de-DE"/
 ><xs:element ref="el"/><xs:element ref="el-GR"/><xs:element ref="en"/><xs:element ref="en-AU"/><
 xs:element ref="en-GB"/><xs:element ref="en-GB-WLS"/><xs:element ref="en-IN"/><xs:element ref="en-
 US"/><xs:element ref="es"/><xs:element ref="es-ES"/><xs:element ref="es-MX"/><xs:element ref="es-
 US"/><xs:element ref="fi"/><xs:element ref="fi-FI"/><xs:element ref="fil"/><xs:element ref="fil-PH
 "/><xs:element ref="fr"/><xs:element ref="fr-CA"/><xs:element ref="fr-FR"/><xs:element ref="gu"/><
 xs:element ref="gu-IN"/><xs:element ref="hi"/><xs:element ref="hi-IN"/><xs:element ref="hu"/><
 xs:element ref="hu-HU"/><xs:element ref="id"/><xs:element ref="id-ID"/><xs:element ref="is"/><
 xs:element ref="is-IS"/><xs:element ref="it"/><xs:element ref="it-IT"/><xs:element ref="ja"/><
 xs:element ref="ja-JP"/><xs:element ref="kn"/><xs:element ref="kn-IN"/><xs:element ref="ko"/><
 xs:element ref="ko-KR"/><xs:element ref="ml"/><xs:element ref="ml-IN"/><xs:element ref="nb"/><
 xs:element ref="nb-NO"/><xs:element ref="nl"/><xs:element ref="nl-NL"/><xs:element ref="pl"/><
 xs:element ref="pl-PL"/><xs:element ref="pt"/><xs:element ref="pt-BR"/><xs:element ref="pt-PT"/><
 xs:element ref="ro"/><xs:element ref="ro-RO"/><xs:element ref="ru"/><xs:element ref="ru-RU"/><
 xs:element ref="sk"/><xs:element ref="sk-SK"/><xs:element ref="sv"/><xs:element ref="sv-SE"/><
 xs:element ref="ta"/><xs:element ref="ta-IN"/><xs:element ref="te"/><xs:element ref="te-IN"/><
 xs:element ref="th"/><xs:element ref="th-TH"/><xs:element ref="tr"/><xs:element ref="tr-TR"/><
 xs:element ref="uk"/><xs:element ref="uk-UA"/><xs:element ref="vi"/><xs:element ref="vi-VN"/><
 xs:element ref="yue"/><xs:element ref="yue-HK"/>

        </xs:choice>
        <xs:attribute name="startmark" type="xs:string" use="optional" />
        <xs:attribute name="endmark" type="xs:string" use="optional" />
    </xs:complexType>
</xs:element>
```

```
</xs:schema>
```

LSML Feature Support Table

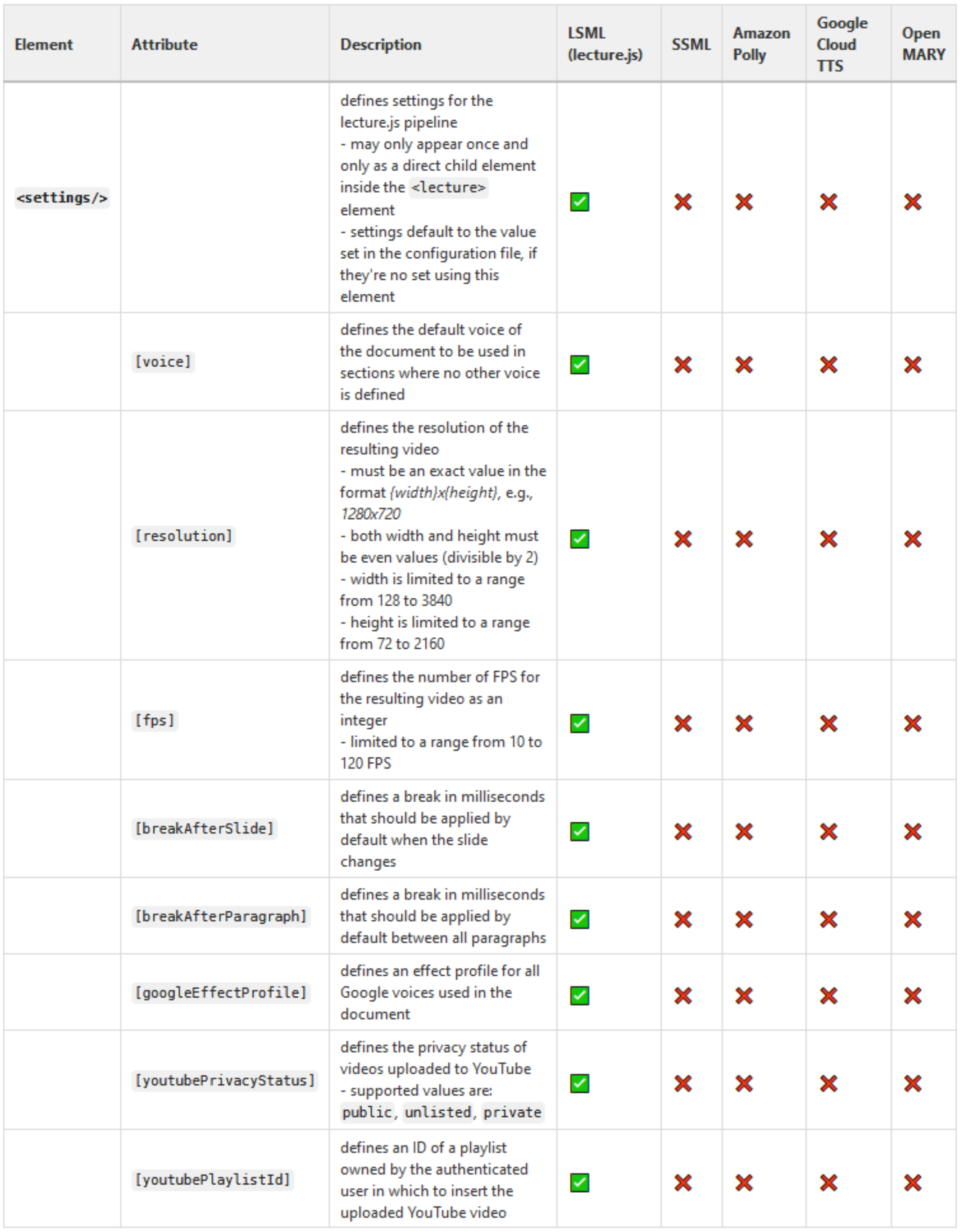
- ✅ means the feature is support
- ❌ means the feature is not supported
- `<element>` is an element with content and `<element/>` an empty element
- `attribute` is a required and `[attribute]` an optional attribute

Document Structure

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<lecture>`		root element of a lecture.js document - the following elements are required as direct children: **1** `<info/>` , **1** `<settings/>`, **1+** `<deck/>`	✅ - replaces the `<speak>` element from SSML	❌	❌	❌	❌
	`[startmark]`	sets at which `<mark/>` rendering starts	✅	❌	❌	❌	❌
	`[endmark]`	sets at which `<mark/>` rendering ends	✅	❌	❌	❌	❌

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<mark>`		defines a marker (position) inside the document	✅	✅	✅ - supported by Polly with timestamps	✅	✅
	`name`	defines the name of the marker - can be targeted using the attributes `startmark` and `endmark` on the `<speak>` and `<lecture>` elements	✅	✅	✅	✅	✅
	`[chapter]`	defines that a new chapter with the given values as the name begins at this marker	✅	❌	❌	❌	❌

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<info/>`		defines information about the lecture - must only appear once and only as a direct child element inside the `<lecture>` element	✅	❌	❌	❌	❌
	`[title]`	defines the title of the lecture	✅	❌	❌	❌	❌
	`[description]`	defines a short description about the lecture	✅	❌	❌	❌	❌
	`[authors]`	defines the name(s) of the author(s) - preferably use a semicolon-separated list of strings	✅	❌	❌	❌	❌
	`[copyright]`	defines the copyright holder(s) and additional copyright information	✅	❌	❌	❌	❌



Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	Open MARY
`<settings/>`		defines settings for the lecture.js pipeline - may only appear once and only as a direct child element inside the `<lecture>` element - settings default to the value set in the configuration file, if they're no set using this element	✅	❌	❌	❌	❌
	`[voice]`	defines the default voice of the document to be used in sections where no other voice is defined	✅	❌	❌	❌	❌
	`[resolution]`	defines the resolution of the resulting video - must be an exact value in the format *{width}x{height}*, e.g., *1280x720* - both width and height must be even values (divisible by 2) - width is limited to a range from 128 to 3840 - height is limited to a range from 72 to 2160	✅	❌	❌	❌	❌
	`[fps]`	defines the number of FPS for the resulting video as an integer - limited to a range from 10 to 120 FPS	✅	❌	❌	❌	❌
	`[breakAfterSlide]`	defines a break in milliseconds that should be applied by default when the slide changes	✅	❌	❌	❌	❌
	`[breakAfterParagraph]`	defines a break in milliseconds that should be applied by default between all paragraphs	✅	❌	❌	❌	❌
	`[googleEffectProfile]`	defines an effect profile for all Google voices used in the document	✅	❌	❌	❌	❌
	`[youtubePrivacyStatus]`	defines the privacy status of videos uploaded to YouTube - supported values are: `public`, `unlisted`, `private`	✅	❌	❌	❌	❌
	`[youtubePlaylistId]`	defines an ID of a playlist owned by the authenticated user in which to insert the uploaded YouTube video	✅	❌	❌	❌	❌

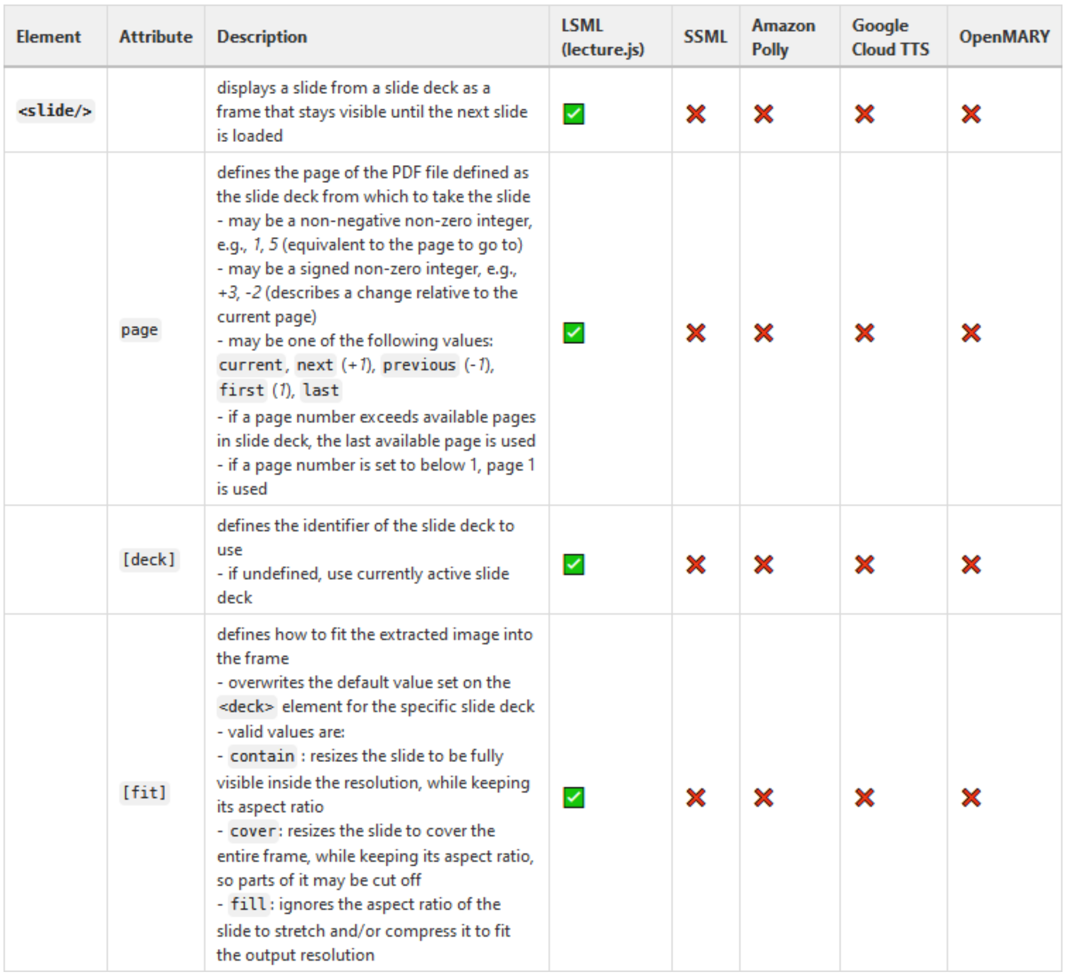
Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<speak>`		root element of a SSML document	✗ - does not support `<speak>`, but instead implements a similar element called `<lecture>`	✓	✓	✓	✓ - converted to `<maryxml>`
	`version`	specifies the version of the SSML specification used	✗	✓	✗	✗	✓
	`xml:lang`	specifies the language of the root document	✗	✓	✗	✗	✓
	`[xml:base]`	specifies the Base URI of the root document	✗	✓	✗	✗	✓
	`[xmlns]`	defines a default namespace for all elements	✗	✓	✗	✗	✓
	`[xmlns:xsi]`	defines a namespace for all elements prefixed with `xsi`	✗	✓	✗	✗	✓
	`[xsi:schemaLocation]`	indicates the location of the SSML schema	✗	✓	✗	✗	✗
	`[onlangfailure]`	specifies the behavior once the speaking language fails	✗	✓	✗	✗	✗
	`[startmark]`	sets at which `<mark/>` rendering starts	✗	✓	✗	✗	✗
	`[endmark]`	sets at which `<mark/>` rendering ends	✗	✓	✗	✗	✗

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<meta/>`		defines meta information about the document - associates a string to a declared meta property or declares `http-equiv` content - must appear only as a direct child of the `<speak>` element and before all content elements	✗	✓	✗	✗	✗
	`name`	declares a custom attribute name *(must not be combined with `http-equiv`)*	✗	✓	✗	✗	✗
	`http-equiv`	declares a custom attribute name with special significance when received via HTTP_(must not be combined with `name`)_	✗	✓	✗	✗	✗
	`content`	provides the content for either `name` or `http-equiv`	✗	✓	✗	✗	✗

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<metadata>`		adds meta information about the document using a metadata schema - any metadata schema is valid, but RDF-XMLSYNTAX is recommended - none of its content is rendered by the speech synthesizer - must appear only as a direct child of the `<speak>` element and before all content elements	✗	✓	✗	✗	✗

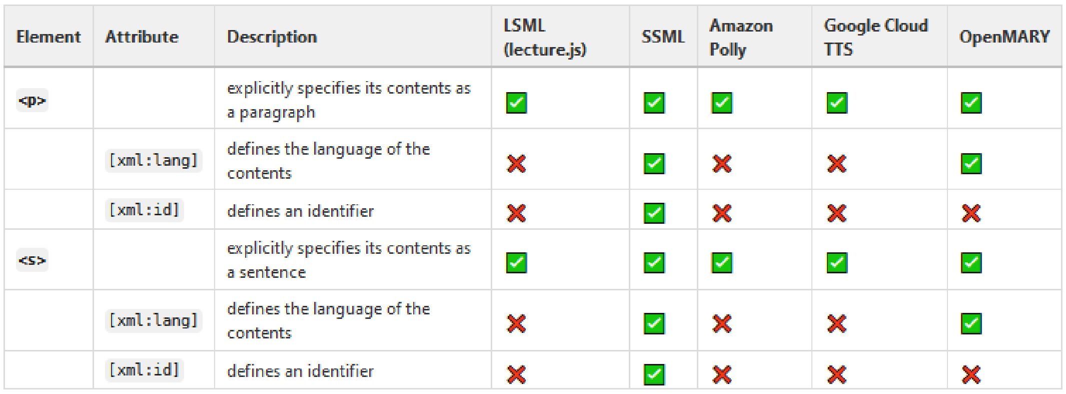
Slide Control

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<deck/>`		loads a PDF file and assigns it a unique identifier - must only appear as a direct child of the `<lecture>` element - multiple slide decks may be loaded, but each needs its own `<deck>` element	✓	✗	✗	✗	✗
	`id`	defines a unique identifier for the slide	✓	✗	✗	✗	✗
	`src`	defines the path to a local PDF file	✓	✗	✗	✗	✗
	`[active]`	sets the deck to be active, which means it's used as long as no other slide deck is loaded using a `<slide>` element - exactly one slide can and must be set to active - can be `true` or `false` (default) - if set to `false`, it's equivalent to not using the `active` attribute	✓	✗	✗	✗	✗
	`[fit]`	defines how to fit the image frames generated from the slide deck into the video - these are default values, which may be overwritten for an individual page using the `<slide>` element - valid values are: - `contain` *(default)*: resizes the slide deck to be fully visible inside the resolution, while keeping its aspect ratio - `cover`: resizes the slide deck to cover the entire frame, while keeping its aspect ratio, so parts of it may be cut off - `fill`: ignores the aspect ratio of the slide deck to stretch and/or compress it to fit the output resolution	✓	✗	✗	✗	✗

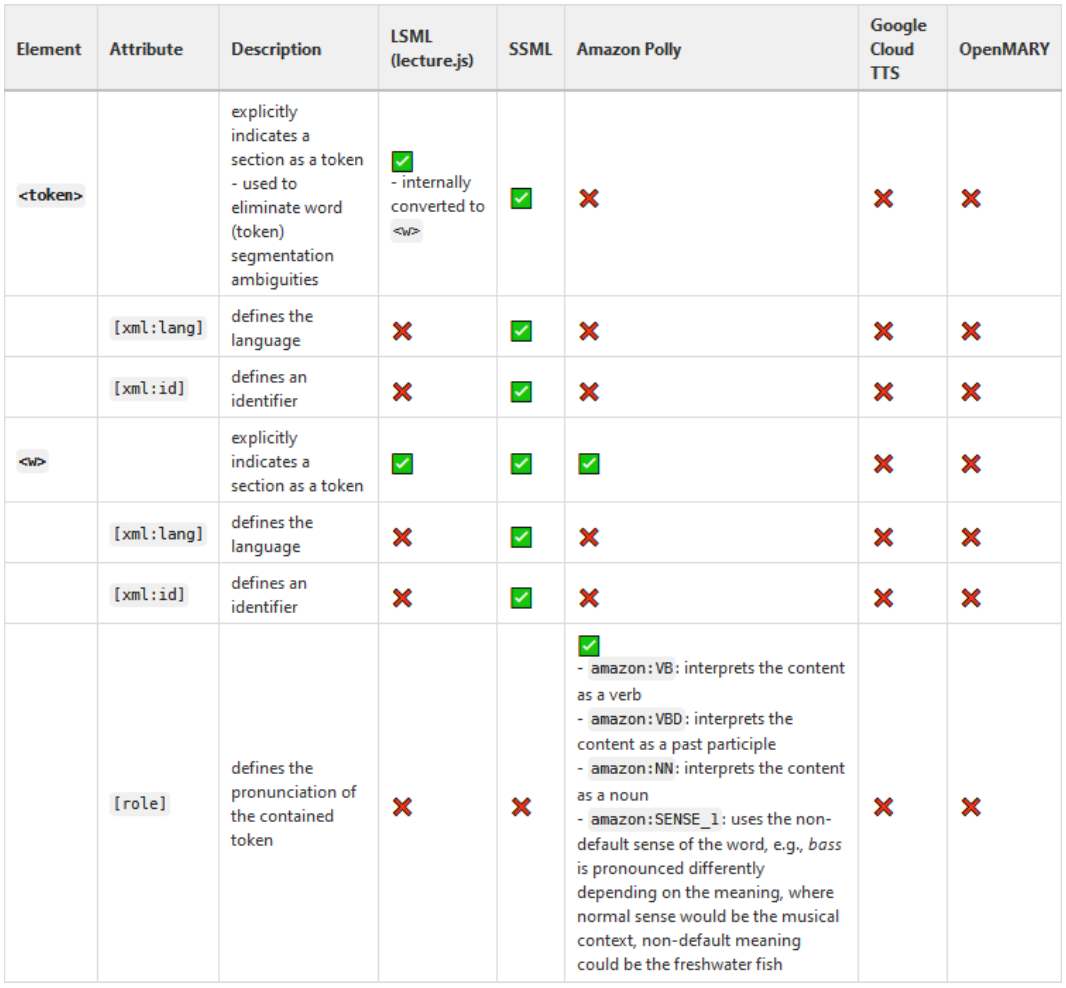


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<slide/>`		displays a slide from a slide deck as a frame that stays visible until the next slide is loaded	✅	❌	❌	❌	❌
	`page`	defines the page of the PDF file defined as the slide deck from which to take the slide - may be a non-negative non-zero integer, e.g., *1*, *5* (equivalent to the page to go to) - may be a signed non-zero integer, e.g., *+3*, *-2* (describes a change relative to the current page) - may be one of the following values: `current`, `next` (*+1*), `previous` (*-1*), `first` (*1*), `last` - if a page number exceeds available pages in slide deck, the last available page is used - if a page number is set to below 1, page 1 is used	✅	❌	❌	❌	❌
	`[deck]`	defines the identifier of the slide deck to use - if undefined, use currently active slide deck	✅	❌	❌	❌	❌
	`[fit]`	defines how to fit the extracted image into the frame - overwrites the default value set on the `<deck>` element for the specific slide deck - valid values are: - `contain` : resizes the slide to be fully visible inside the resolution, while keeping its aspect ratio - `cover`: resizes the slide to cover the entire frame, while keeping its aspect ratio, so parts of it may be cut off - `fill`: ignores the aspect ratio of the slide to stretch and/or compress it to fit the output resolution	✅	❌	❌	❌	❌

Text Structure

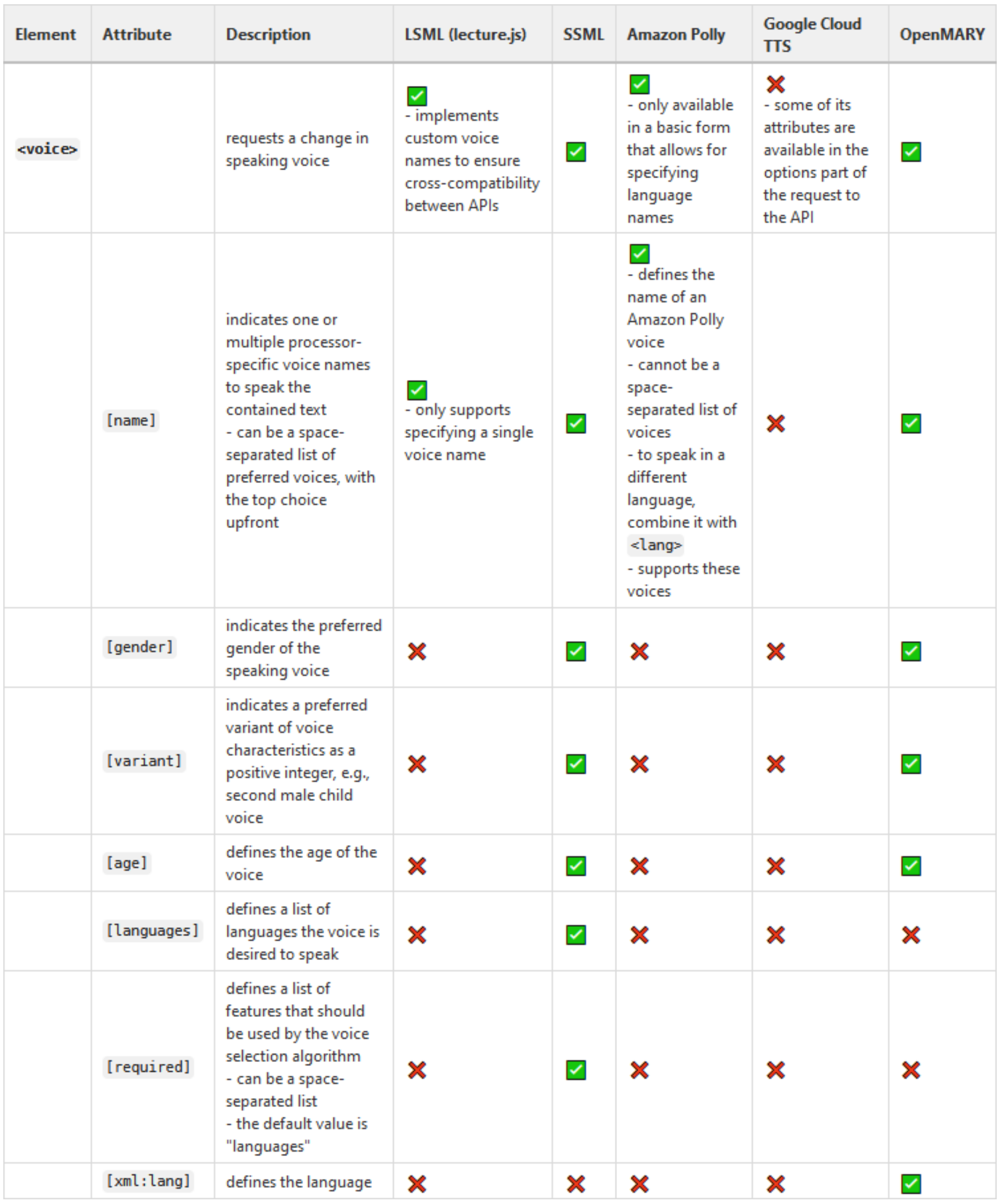


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<p>`		explicitly specifies its contents as a paragraph	✅	✅	✅	✅	✅
	`[xml:lang]`	defines the language of the contents	❌	✅	❌	❌	✅
	`[xml:id]`	defines an identifier	❌	✅	❌	❌	❌
`<s>`		explicitly specifies its contents as a sentence	✅	✅	✅	✅	✅
	`[xml:lang]`	defines the language of the contents	❌	✅	❌	❌	✅
	`[xml:id]`	defines an identifier	❌	✅	❌	❌	❌

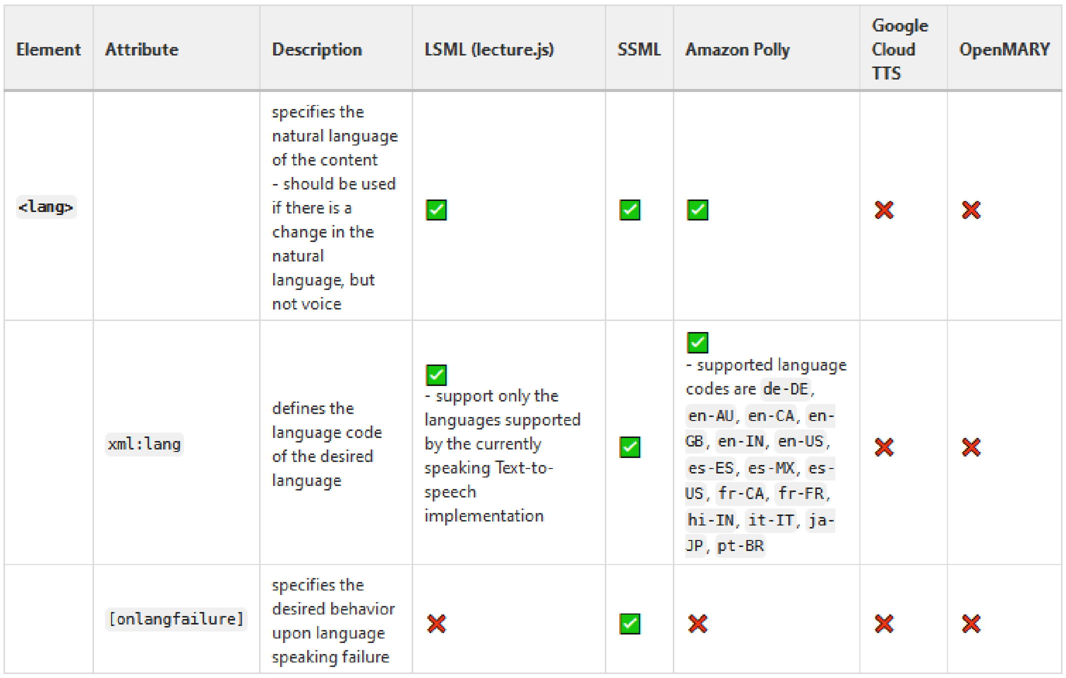


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<token>`		explicitly indicates a section as a token - used to eliminate word (token) segmentation ambiguities	✅ - internally converted to `<w>`	✅	❌	❌	❌
	`[xml:lang]`	defines the language	❌	✅	❌	❌	❌
	`[xml:id]`	defines an identifier	❌	✅	❌	❌	❌
`<w>`		explicitly indicates a section as a token	✅	✅	✅	❌	❌
	`[xml:lang]`	defines the language	❌	✅	❌	❌	❌
	`[xml:id]`	defines an identifier	❌	✅	❌	❌	❌
	`[role]`	defines the pronunciation of the contained token	❌	❌	✅ - `amazon:VB`: interprets the content as a verb - `amazon:VBD`: interprets the content as a past participle - `amazon:NN`: interprets the content as a noun - `amazon:SENSE_1`: uses the non-default sense of the word, e.g., *bass* is pronounced differently depending on the meaning, where normal sense would be the musical context, non-default meaning could be the freshwater fish	❌	❌

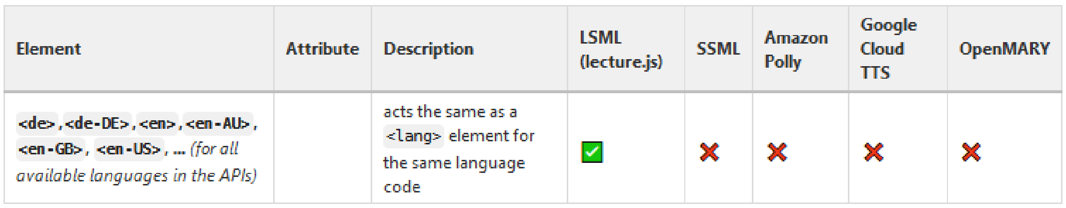
Voices and Languages



Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<voice>`		requests a change in speaking voice	✅ - implements custom voice names to ensure cross-compatibility between APIs	✅	✅ - only available in a basic form that allows for specifying language names	❌ - some of its attributes are available in the options part of the request to the API	✅
	`[name]`	indicates one or multiple processor-specific voice names to speak the contained text - can be a space-separated list of preferred voices, with the top choice upfront	✅ - only supports specifying a single voice name	✅	✅ - defines the name of an Amazon Polly voice - cannot be a space-separated list of voices - to speak in a different language, combine it with `<lang>` - supports these voices	❌	✅
	`[gender]`	indicates the preferred gender of the speaking voice	❌	✅	❌	❌	✅
	`[variant]`	indicates a preferred variant of voice characteristics as a positive integer, e.g., second male child voice	❌	✅	❌	❌	✅
	`[age]`	defines the age of the voice	❌	✅	❌	❌	✅
	`[languages]`	defines a list of languages the voice is desired to speak	❌	✅	❌	❌	❌
	`[required]`	defines a list of features that should be used by the voice selection algorithm - can be a space-separated list - the default value is "languages"	❌	✅	❌	❌	❌
	`[xml:lang]`	defines the language	❌	❌	❌	❌	✅

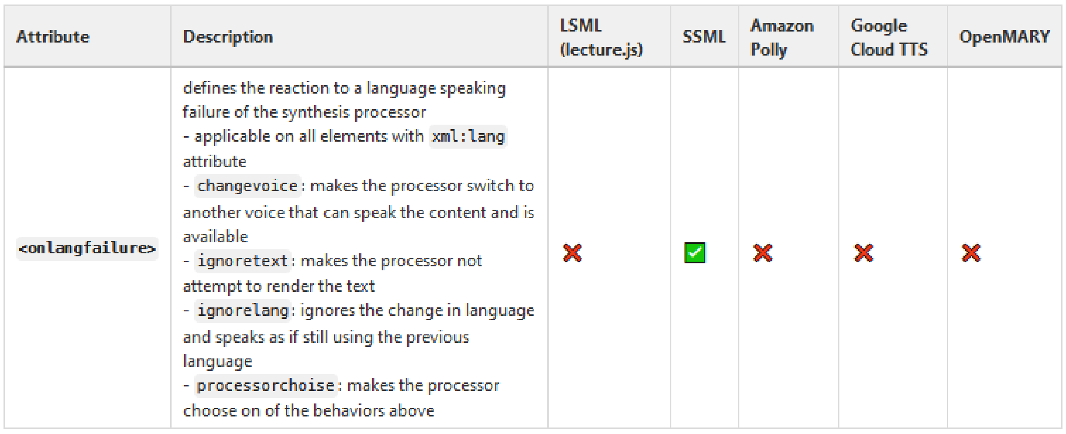


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<lang>`		specifies the natural language of the content - should be used if there is a change in the natural language, but not voice	✅	✅	✅	❌	❌
	`xml:lang`	defines the language code of the desired language	✅ - support only the languages supported by the currently speaking Text-to-speech implementation	✅	✅ - supported language codes are `de-DE`, `en-AU`, `en-CA`, `en-GB`, `en-IN`, `en-US`, `es-ES`, `es-MX`, `es-US`, `fr-CA`, `fr-FR`, `hi-IN`, `it-IT`, `ja-JP`, `pt-BR`	❌	❌
	`[onlangfailure]`	specifies the desired behavior upon language speaking failure	❌	✅	❌	❌	❌



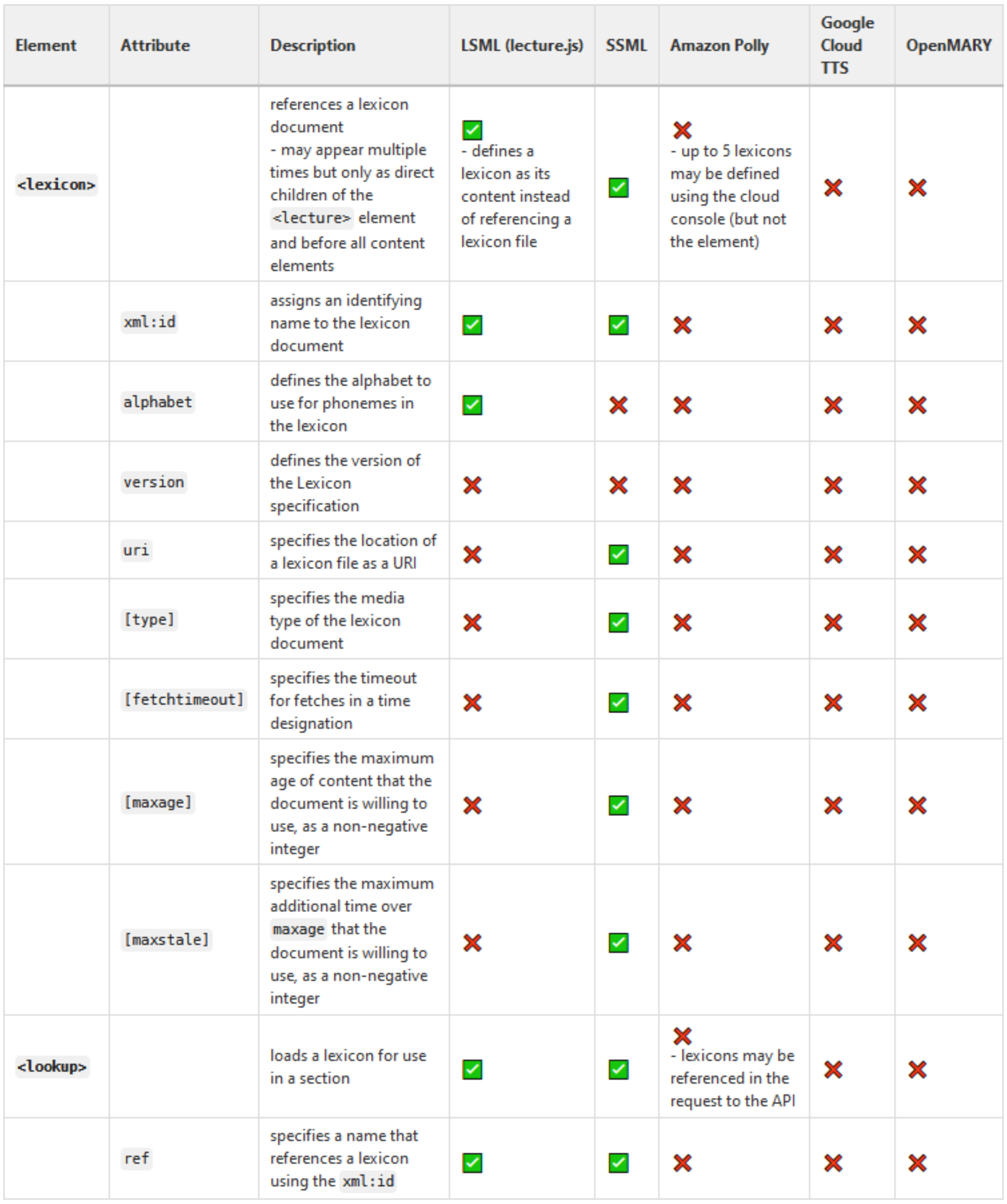
Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<de>`, `<de-DE>`, `<en>`, `<en-AU>`, `<en-GB>`, `<en-US>`, ... *(for all available languages in the APIs)*		acts the same as a `<lang>` element for the same language code	✅	❌	❌	❌	❌

Generic Attributes



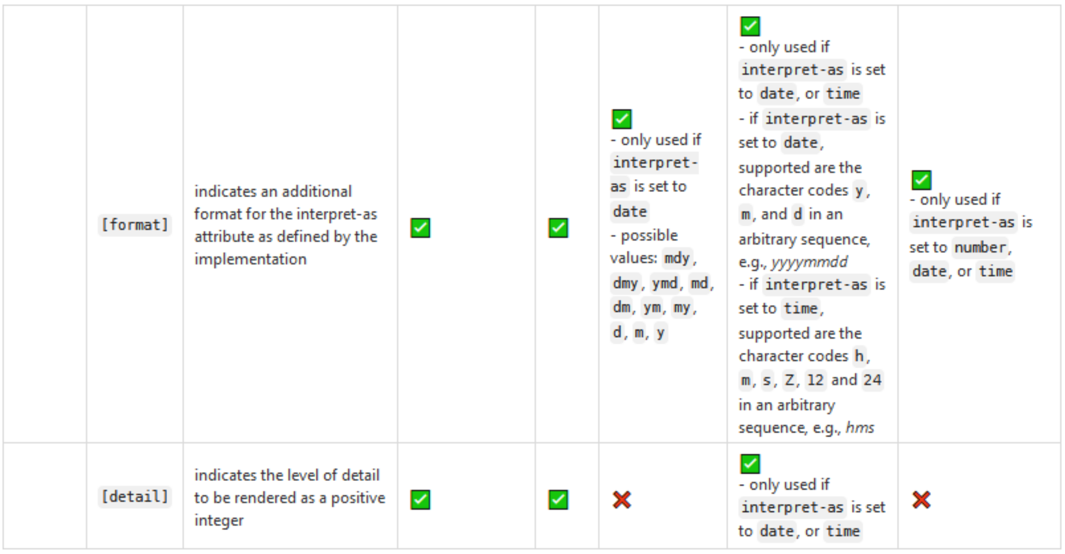
Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<onlangfailure>`	defines the reaction to a language speaking failure of the synthesis processor - applicable on all elements with `xml:lang` attribute - `changevoice`: makes the processor switch to another voice that can speak the content and is available - `ignoretext`: makes the processor not attempt to render the text - `ignorelang`: ignores the change in language and speaks as if still using the previous language - `processorchoise`: makes the processor choose on of the behaviors above	❌	✅	❌	❌	❌

Correcting Mispronunciations

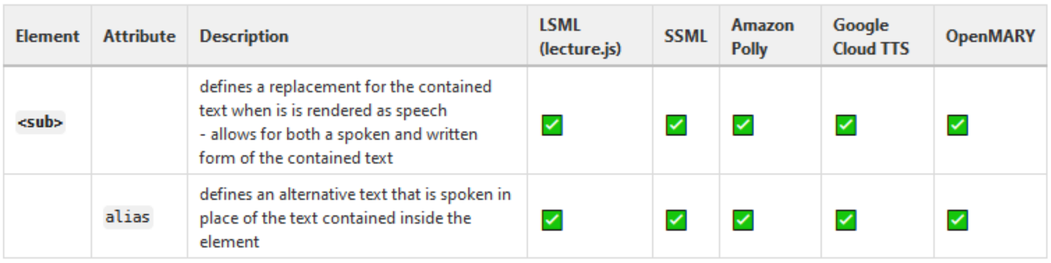


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<lexicon>`		references a lexicon document - may appear multiple times but only as direct children of the `<lecture>` element and before all content elements	✓ - defines a lexicon as its content instead of referencing a lexicon file	✓	✗ - up to 5 lexicons may be defined using the cloud console (but not the element)	✗	✗
	`xml:id`	assigns an identifying name to the lexicon document	✓	✓	✗	✗	✗
	`alphabet`	defines the alphabet to use for phonemes in the lexicon	✓	✗	✗	✗	✗
	`version`	defines the version of the Lexicon specification	✗	✗	✗	✗	✗
	`uri`	specifies the location of a lexicon file as a URI	✗	✓	✗	✗	✗
	`[type]`	specifies the media type of the lexicon document	✗	✓	✗	✗	✗
	`[fetchtimeout]`	specifies the timeout for fetches in a time designation	✗	✓	✗	✗	✗
	`[maxage]`	specifies the maximum age of content that the document is willing to use, as a non-negative integer	✗	✓	✗	✗	✗
	`[maxstale]`	specifies the maximum additional time over `maxage` that the document is willing to use, as a non-negative integer	✗	✓	✗	✗	✗
`<lookup>`		loads a lexicon for use in a section	✓	✓	✗ - lexicons may be referenced in the request to the API	✗	✗
	`ref`	specifies a name that references a lexicon using the `xml:id`	✓	✓	✗	✗	✗

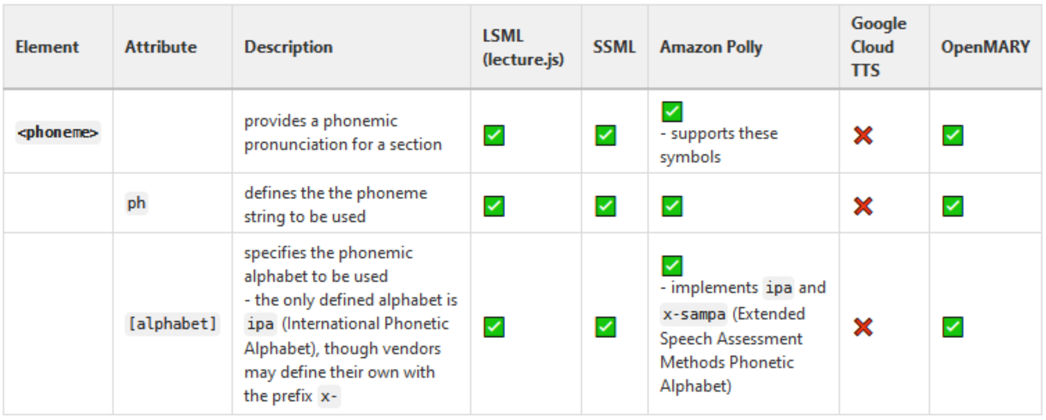
Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<say-as>`		describes how the text should be interpreted by providing additional context - the attribute names are set by SSML specification, but the available values are arbitrary set by the different speech synthesis APIs	✓	✓	✓	✓	✓
	`interpret-as`	indicates the content type of the contained text construct - `cardinal` / `number`: interprets the values as a cardinal number, e.g., *123* as *One hundred twenty-three* - `ordinal`: interprets the value as an ordinal number, e.g., *1* as *First* - `characters` / `verbatim` / `spell-out`: spells out each letter - `fraction`: interprets the value as a fraction, e.g., *1+1/2* as *one and a half* - `expletive` / `bleep`: bleeps out the content inside the element as if censored - `unit`: interprets the value as a measurement, e.g., *10kg* as *ten kilogram* - `date`: interprets the value as a date of which the format can be defined in `format`, or detail level using `detail` - `time`: interprets the value as time, e.g., *1'21"* as a duration in minutes and seconds or *1:20pm* as a time of day, of which the format may be set using `format` - `telephone`: interprets the value as a 7 or 10 digit telephone number (W3 Specification) - `address`: interprets the value as part of a street address - `interjection`: interprets the value as an interjection (speaks the text more expressively)	✓ - may only contain text - only supports types supported by Amazon Polly and Google Cloud TTS, which are: - `cardinal` - `ordinal` - `characters` - `spell-out` - `fraction` - `expletive` - `unit` - `date` - `time` - `telephone`	✓	✓ - does not support the synonyms `bleep`, `verbatim` - only supports `interjection` for certain speechcons	✓ - does not support the synonym `number` - does not support the values `address`, `interjection` - `unit` also converts units to a singular or plural depending on the number - `characters` may spell out a string boldly, for example *can* as *C A N*, in contrast to `verbatim` and `spell-out`	✓ - does not support the synonyms `verbatim` / `spell-out` (even though `spell-out` is used internally) - does not support the values `expletive` / `bleep`, `unit`, `fraction`, `address`, `interjection`



	`[format]`	indicates an additional format for the interpret-as attribute as defined by the implementation	✓	✓	✓ - only used if `interpret-as` is set to `date` - possible values: `mdy`, `dmy`, `ymd`, `md`, `dm`, `ym`, `my`, `d`, `m`, `y`	✓ - only used if `interpret-as` is set to `date`, or `time` - if `interpret-as` is set to `date`, supported are the character codes `y`, `m`, and `d` in an arbitrary sequence, e.g., *yyyymmdd* - if `interpret-as` is set to `time`, supported are the character codes `h`, `m`, `s`, `Z`, `12` and `24` in an arbitrary sequence, e.g., *hms*	✓ - only used if `interpret-as` is set to `number`, `date`, or `time`
	`[detail]`	indicates the level of detail to be rendered as a positive integer	✓	✓	✗	✓ - only used if `interpret-as` is set to `date`, or `time`	✗

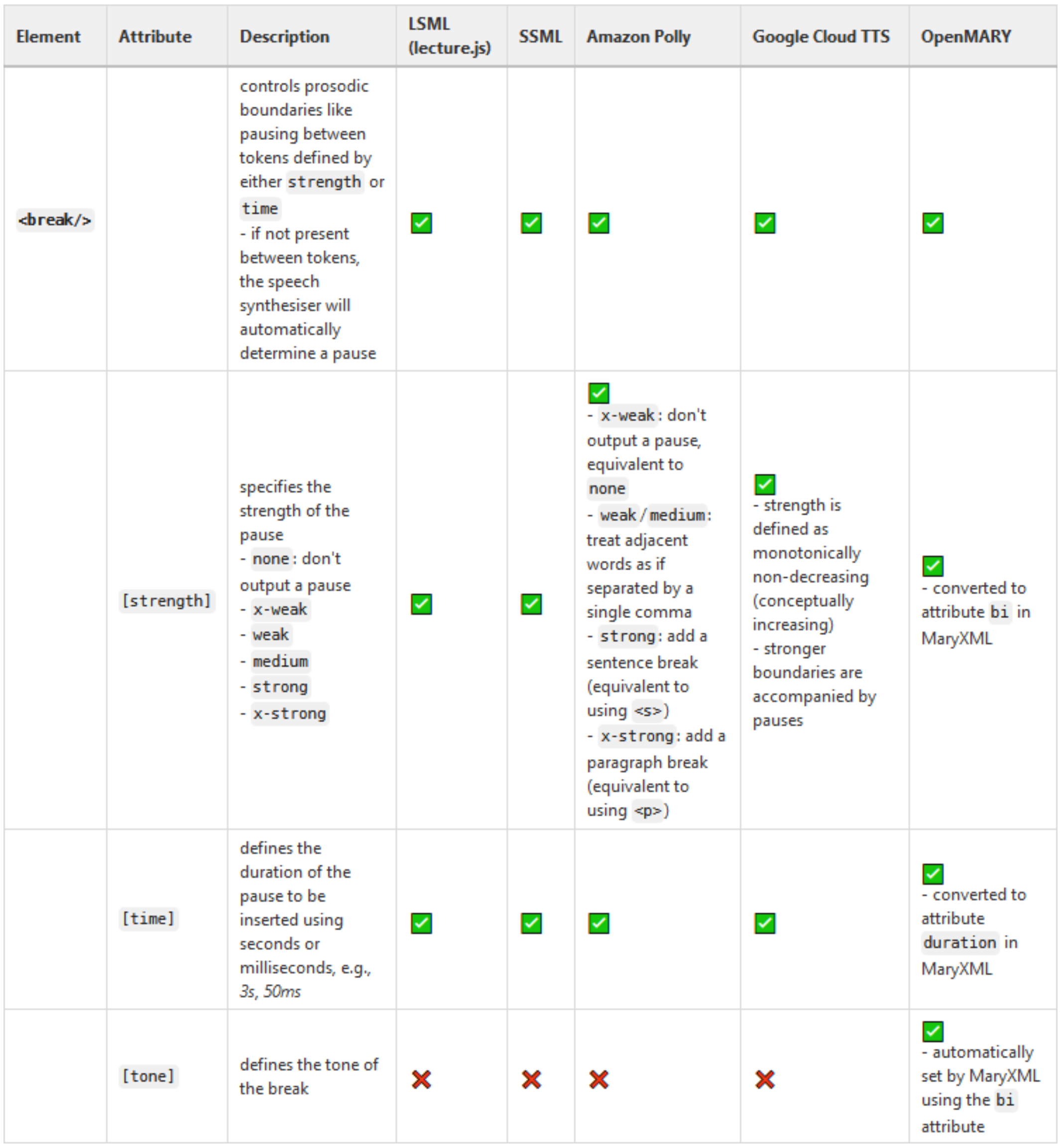


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<sub>`		defines a replacement for the contained text when is is rendered as speech - allows for both a spoken and written form of the contained text	✓	✓	✓	✓	✓
	`alias`	defines an alternative text that is spoken in place of the text contained inside the element	✓	✓	✓	✓	✓

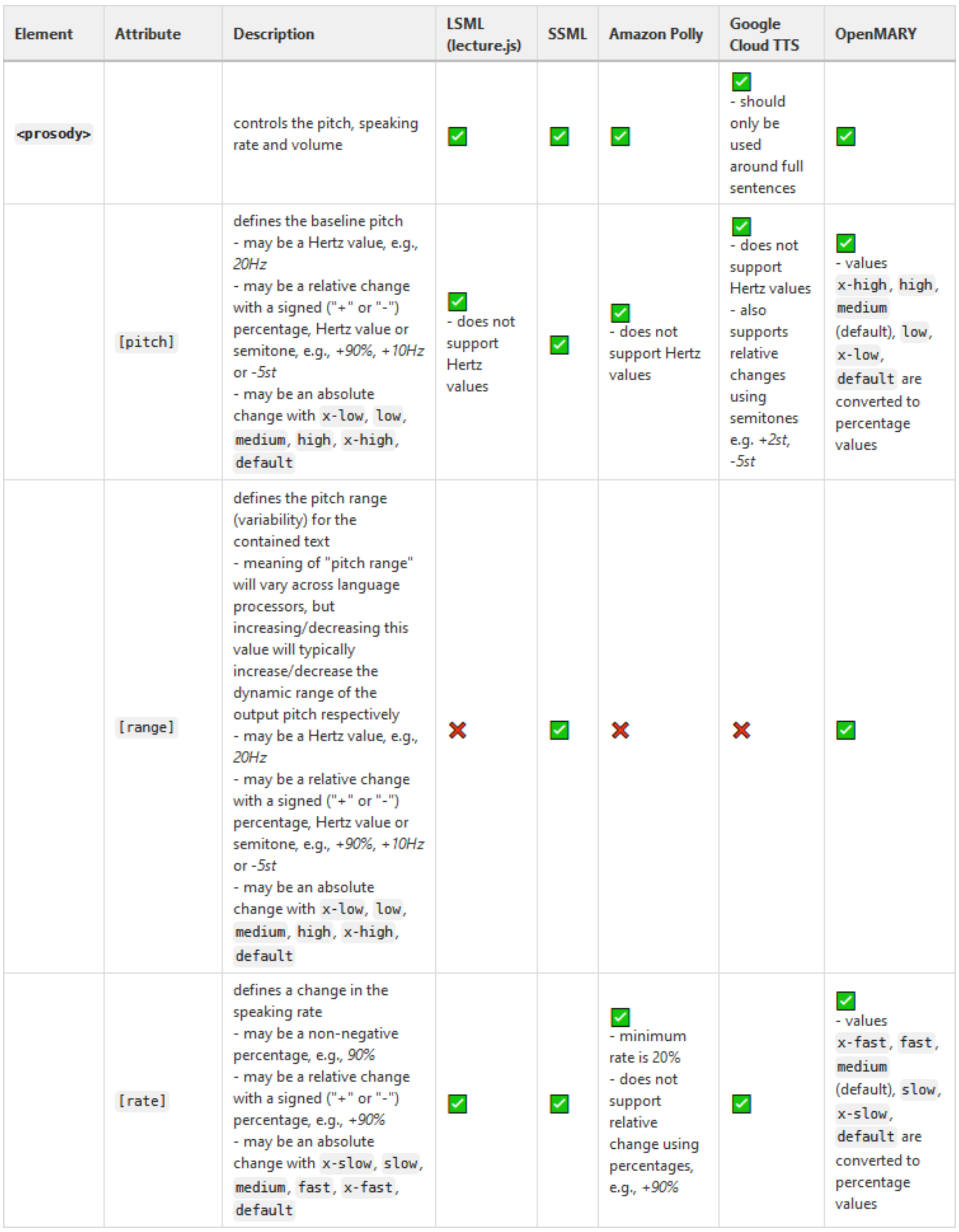


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<phoneme>`		provides a phonemic pronunciation for a section	✓	✓	✓ - supports these symbols	✗	✓
	`ph`	defines the the phoneme string to be used	✓	✓	✓	✗	✓
	`[alphabet]`	specifies the phonemic alphabet to be used - the only defined alphabet is `ipa` (International Phonetic Alphabet), though vendors may define their own with the prefix `x-`	✓	✓	✓ - implements `ipa` and `x-sampa` (Extended Speech Assessment Methods Phonetic Alphabet)	✗	✓

Prosodic Features



Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<break/>`		controls prosodic boundaries like pausing between tokens defined by either `strength` or `time` - if not present between tokens, the speech synthesiser will automatically determine a pause	✓	✓	✓	✓	✓
	`[strength]`	specifies the strength of the pause - `none`: don't output a pause - `x-weak` - `weak` - `medium` - `strong` - `x-strong`	✓	✓	✓ - `x-weak`: don't output a pause, equivalent to `none` - `weak` / `medium`: treat adjacent words as if separated by a single comma - `strong`: add a sentence break (equivalent to using `<s>`) - `x-strong`: add a paragraph break (equivalent to using `<p>`)	✓ - strength is defined as monotonically non-decreasing (conceptually increasing) - stronger boundaries are accompanied by pauses	✓ - converted to attribute `bi` in MaryXML
	`[time]`	defines the duration of the pause to be inserted using seconds or milliseconds, e.g., *3s*, *50ms*	✓	✓	✓	✓	✓ - converted to attribute `duration` in MaryXML
	`[tone]`	defines the tone of the break	✗	✗	✗	✗	✓ - automatically set by MaryXML using the `bi` attribute

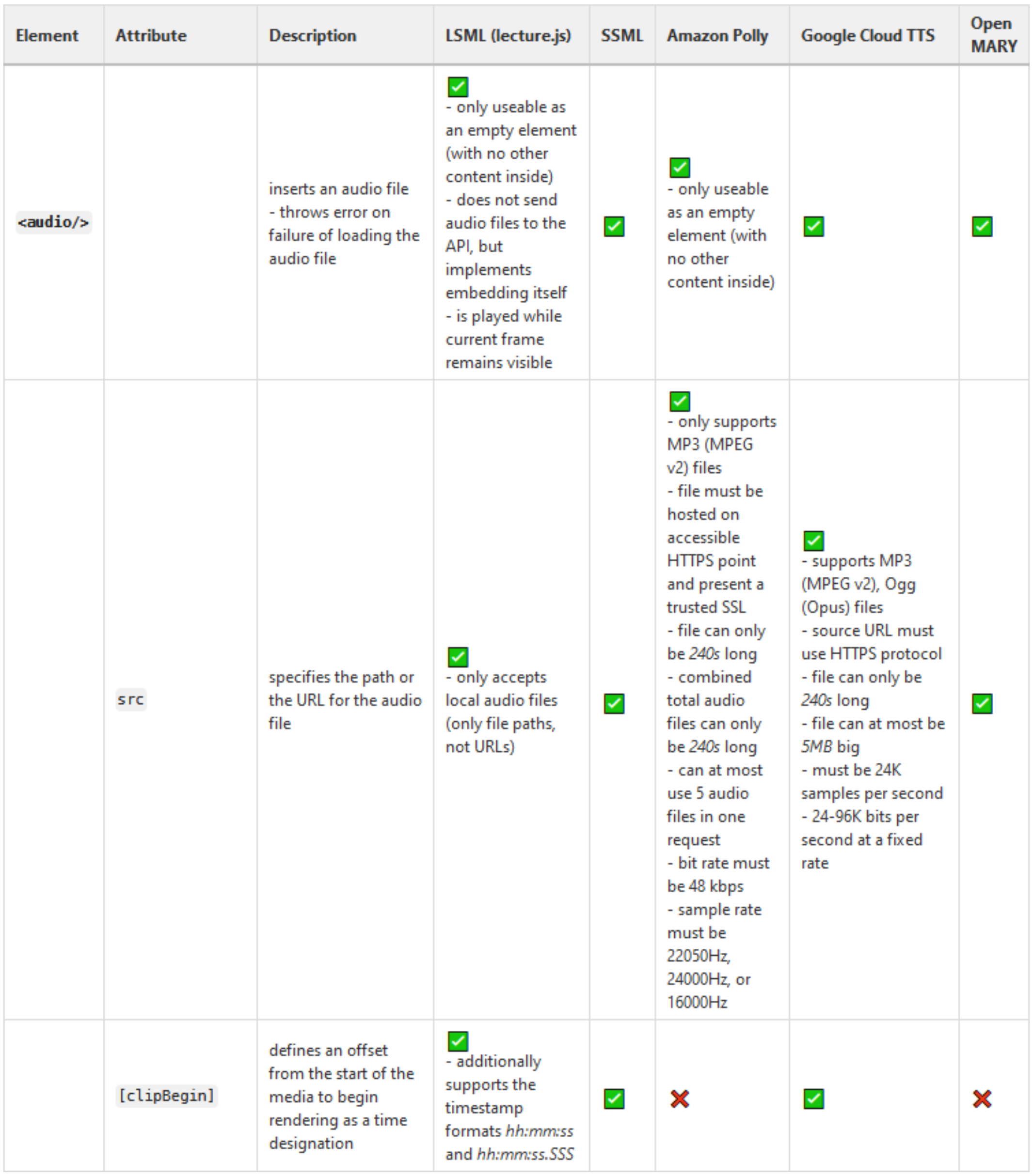


Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<prosody>`		controls the pitch, speaking rate and volume	✓	✓	✓	✓ - should only be used around full sentences	✓
	`[pitch]`	defines the baseline pitch - may be a Hertz value, e.g., *20Hz* - may be a relative change with a signed ("+" or "-") percentage, Hertz value or semitone, e.g., *+90%*, *+10Hz* or *-5st* - may be an absolute change with `x-low`, `low`, `medium`, `high`, `x-high`, `default`	✓ - does not support Hertz values	✓	✓ - does not support Hertz values	✓ - does not support Hertz values - also supports relative changes using semitones e.g. *+2st*, *-5st*	✓ - values `x-high`, `high`, `medium` (default), `low`, `x-low`, `default` are converted to percentage values
	`[range]`	defines the pitch range (variability) for the contained text - meaning of "pitch range" will vary across language processors, but increasing/decreasing this value will typically increase/decrease the dynamic range of the output pitch respectively - may be a Hertz value, e.g., *20Hz* - may be a relative change with a signed ("+" or "-") percentage, Hertz value or semitone, e.g., *+90%*, *+10Hz* or *-5st* - may be an absolute change with `x-low`, `low`, `medium`, `high`, `x-high`, `default`	✗	✓	✗	✗	✓
	`[rate]`	defines a change in the speaking rate - may be a non-negative percentage, e.g., *90%* - may be a relative change with a signed ("+" or "-") percentage, e.g., *+90%* - may be an absolute change with `x-slow`, `slow`, `medium`, `fast`, `x-fast`, `default`	✓	✓	✓ - minimum rate is 20% - does not support relative change using percentages, e.g., *+90%*	✓	✓ - values `x-fast`, `fast`, `medium` (default), `slow`, `x-slow`, `default` are converted to percentage values

	`[volume]`	defines the volume for the contained text - may be a signed ("+" or "-") number followed by "dB", e.g., *+0.5dB* - may be a relative change with a signed ("+" or "-") percentage, e.g., *+90%* - may be an absolute value like `silent`, `x-soft`, `soft`, `medium`, `loud`, `x-loud`, `default`	✅	✅	✅ - "-6dB" is roughly half the current amplitude - maximum positive dB value is *+4.08dB* - does not support relative change using percentages, e.g., *+90%*	✅	✅ - values `x-loud`, `loud`, `medium` (default), `soft`, `x-soft`, `silent`, `default` are converted to percentage values
	`amazon:max-duration`	defines the maximum duration in seconds or milliseconds	❌	❌	✅ - not available with Neural Voices	❌	❌
	`[duration]`	defines the desired time for the narrator to read the text - takes precedence over the `rate` attribute - may be a time declaration following the CSS2 Recommendation, e.g., "2s", "50ms" - may be a relative change with a signed ("+" or "-") percentage, e.g., "+90%"	❌	✅	❌	❌	❌
	`[contour]`	defines an interpolation of different pitch values on the text within - defined as a set of white space-separated targets at specified time positions in the speech output	❌	✅	❌	❌	✅

Element	**Attribute**	**Description**	**LSML (lecture.js)**	**SSML**	**Amazon Polly**	**Google Cloud TTS**	**OpenMARY**
`<emphasis>`		requests the contained text to be spoken with a specific level of emphasis - effect may differ between languages, dialects, voices and APIs	✅	✅	✅ - does not alter the pronunciation of words and only changes rate and volume - not available with Neural Voices	✅	✅
	`level`	indicates the strength of emphasis to be applied - valid values are: - `none`: does not add an emphasis effect - `strong` - `moderate` - `reduced`	✅	✅	✅ - changes rate and volume, e.g., `strong` is louder and slower - does not support the `none` level	✅	✅ - is converted to pitch and rate levels in MaryXML

External resources



Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	Open MARY
`<audio/>`		inserts an audio file - throws error on failure of loading the audio file	✅ - only useable as an empty element (with no other content inside) - does not send audio files to the API, but implements embedding itself - is played while current frame remains visible	✅	✅ - only useable as an empty element (with no other content inside)	✅	✅
	`src`	specifies the path or the URL for the audio file	✅ - only accepts local audio files (only file paths, not URLs)	✅	✅ - only supports MP3 (MPEG v2) files - file must be hosted on accessible HTTPS point and present a trusted SSL - file can only be *240s* long - combined total audio files can only be *240s* long - can at most use 5 audio files in one request - bit rate must be 48 kbps - sample rate must be 22050Hz, 24000Hz, or 16000Hz	✅ - supports MP3 (MPEG v2), Ogg (Opus) files - source URL must use HTTPS protocol - file can only be *240s* long - file can at most be *5MB* big - must be 24K samples per second - 24-96K bits per second at a fixed rate	✅
	`[clipBegin]`	defines an offset from the start of the media to begin rendering as a time designation	✅ - additionally supports the timestamp formats *hh:mm:ss* and *hh:mm:ss.SSS*	✅	❌	✅	❌

	`[clipEnd]`	defines an offset from the start of the media to end rendering as a time designation	✓ - additionally supports the timestamp formats *hh:mm:ss* and *hh:mm:ss.SSS*	✓	✗	✓	✗
	`[soundLevel]`	defines the relative volume of the referenced audio - takes signed ("+" or "-") CSS2 numbers followed by "dB"	✓ - limited to integer values between *+50dB* and *-50dB*	✓	✗	✓ - maximum range is -/+40dB, but effective range may be smaller	✗
	`[speed]`	defines the playback speed of the referenced audio in percentage - takes a positive real percentage value, e.g., "90%"	✓ - limited to integer values between 50% and 200%	✓	✗	✓ - allowed range is 50-200% - values outside that range may be adjusted to be within range	✗
	`[repeatCount]`	defines how often to loop the audio as a positive integer	✓	✓	✗	✓	✗
	`[repeatDur]`	defines the total duration of repeatedly rendering the media	✗	✓	✗	✓ - limits duration after `clipBegin`, `clipEnd`, `repeatCount` and `speed` have been applied	✗
	`[fetchtimeout]`	specifies the timeout for fetches as a time designation	✗	✓	✗	✗	✗
	`[fetchhint]`	tells the synthesis processor if it may pre-fetch audio - `safe` - `prefetch`	✗	✓	✗	✗	✗
	`[maxage]`	indicates the maximum age of content that the document is willing to use, as a non-negative integer	✗	✓	✗	✗	✗
	`[maxstale]`	indicates the maximum additional time over `maxage` of content that the document is willing to use, as a non-negative integer	✗	✓	✗	✗	✗

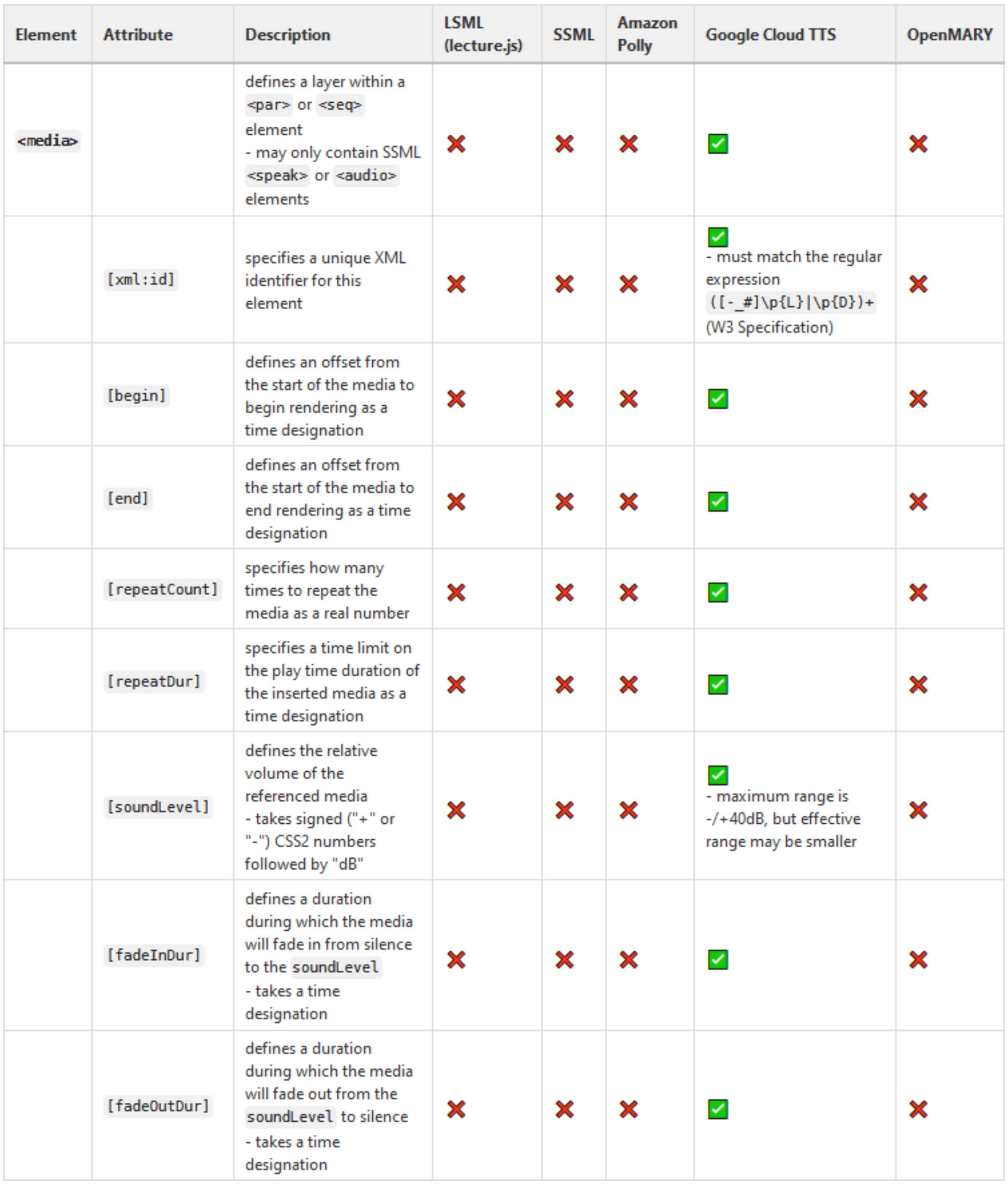
Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<desc>`		adds a description to the audio file, e.g., describing sound effects like "*door slamming*" - optional element that may only occur inside of the `<audio>` element	✗	✓	✗	✓ - contents are rendered as speech if the audio file cannot be played	✗

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<video/>`		inserts a video file to be played - throws error on failure of finding the file	✅	❌	❌	❌	❌
	`src`	defines the path to a local video file	✅	❌	❌	❌	❌
	`[clipBegin]`	defines an offset when to start rendering the video relative to the beginning of the clip, e.g., *2s*, *2000ms*, *hh:mm:ss.SS*, *hh:mm:ss*	✅	❌	❌	❌	❌
	`[clipEnd]`	defines an offset when to stop rendering the video relative to the beginning of the clip, e.g., *2s*, *2000ms*, *hh:mm:ss.SS*, *hh:mm:ss*	✅	❌	❌	❌	❌
	`[keepFrame]`	defines if after the video was played, the last frame should remain as the current video lecture frame, or if the lecture should display the last slide again	✅	❌	❌	❌	❌
	`[fit]`	defines how to fit the video into the frame - valid values are: - `contain` *(default)*: resizes the video to be fully visible inside the resolution, while keeping its aspect ratio - `cover`: resizes the video to cover the entire frame, while keeping its aspect ratio, so parts of it may be cut off - `fill`: ignores the aspect ratio of the video to stretch and/or compress it to fit the output resolution	✅	❌	❌	❌	❌
	`[soundLevel]`	defines the relative volume of the audio channel of the referenced video file - takes signed ("+" or "-") CSS2 numbers followed by "dB"	✅ - limited to values between *+50dB* and *-50dB*	❌	❌	❌	❌
	`[speed]`	defines the playback speed of the referenced video file in percentage - takes a positive real percentage value, e.g., *90%*	✅ - limited to values between *50%* and *200%*	❌	❌	❌	❌
	`[repeatCount]`	defines how often to loop the video as a positive integer	✅	❌	❌	❌	❌

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<image/>`		inserts a image file to be shown as the backdrop, while the voice speaks, until a `<slide>`, `<video>` or different `<image>` element is used again - throws error on failure of finding the file	✅	❌	❌	❌	❌
	`src`	defines the path to a local image file	✅	❌	❌	❌	❌
	`[fit]`	defines how to fit the image into the frame - valid values are: - `contain` *(default)*: resizes the image to be fully visible inside the resolution, while keeping its aspect ratio - `cover`: resizes the image to cover the entire frame, while keeping its aspect ratio, so parts of it may be cut off - `fill`: ignores the aspect ratio of the image to stretch and/or compress it to fit the output resolution	✅	❌	❌	❌	❌

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<seq>`		element that allows to play media elements one after the other - can only contain `<seq>`, `<par>` and `<media>` elements - the beginning and end child elements can have offsets	❌	❌	❌	✅	❌

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<par>`		element that allows play media elements parallel to each other - can only contain `<seq>`, `<par>` and `<media>` elements - child elements take the beginning time of the `<par>` container - `begin` attribute is ignored on the `par` container, time is set at the location of the element in the text - if child `<media>` elements define a `begin` or `end` time, the offset is relative to the `<par>` container	❌	❌	❌	✅	❌



Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<media>`		defines a layer within a `<par>` or `<seq>` element - may only contain SSML `<speak>` or `<audio>` elements	✗	✗	✗	✓	✗
	`[xml:id]`	specifies a unique XML identifier for this element	✗	✗	✗	✓ - must match the regular expression `([-_#]\p{L}|\p{D})+` (W3 Specification)	✗
	`[begin]`	defines an offset from the start of the media to begin rendering as a time designation	✗	✗	✗	✓	✗
	`[end]`	defines an offset from the start of the media to end rendering as a time designation	✗	✗	✗	✓	✗
	`[repeatCount]`	specifies how many times to repeat the media as a real number	✗	✗	✗	✓	✗
	`[repeatDur]`	specifies a time limit on the play time duration of the inserted media as a time designation	✗	✗	✗	✓	✗
	`[soundLevel]`	defines the relative volume of the referenced media - takes signed ("+" or "-") CSS2 numbers followed by "dB"	✗	✗	✗	✓ - maximum range is -/+40dB, but effective range may be smaller	✗
	`[fadeInDur]`	defines a duration during which the media will fade in from silence to the `soundLevel` - takes a time designation	✗	✗	✗	✓	✗
	`[fadeOutDur]`	defines a duration during which the media will fade out from the `soundLevel` to silence - takes a time designation	✗	✗	✗	✓	✗

Amazon Polly Effects

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<amazon:auto-breath/>`		adds a single breathing sound	✗	✗	✓ - only supported by Polly	✗	✗
	`duration`	controls the length of the breath - valid values are `default`, `x-short`, `short`, `medium` (default), `long`, `x-long`	✗	✗	✓	✗	✗
	`volume`	controls the loudness of the breather - valid values are `default`, `x-soft`, `soft`, `medium` (default), `loud`, `x-loud`	✗	✗	✓	✗	✗

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<amazon:auto-breaths>`		adds breathings sounds automatically at appropriate intervals to the contained text	✗	✗	✓	✗	✗
	`duration`	controls the length of the breaths - valid values are `default`, `x-short`, `short`, `medium` (default), `long`, `x-long`	✗	✗	✓	✗	✗
	`frequency`	controls how often breathing sounds occur in the text - valid values are `default`, `x-low`, `low`, `medium` (default), `high`, `x-high`	✗	✗	✓	✗	✗
	`volume`	controls the loudness of the breathers - valid values are `default`, `x-soft`, `soft`, `medium` (default), `loud`, `x-loud`	✗	✗	✓	✗	✗

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<amazon:emotion>`		makes the voice express a specific emotion	✗	✗	✓	✗	✗
	`name`	defines the name of the specific emotion to apply to the voice - `excited` - `disappointed`	✗	✗	✓	✗	✗
	`intensity`	defines the intensity of the specific emotion - `low` - `medium` - `high`	✗	✗	✓	✗	✗

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<amazon:domain>`		applies different speaking styles to the speech	❌	❌	✅	❌	❌
	`name`	defines the name of the specific speaking style to apply to the voice - `conversational`: sound less formal, more as if talking to friends or family *(requires the Matthew or Joanna voice)* - `long-form`: style suitable for long-form content like podcasts, blogs, articles *(only available in English (US) and not compatible with `<voice>`)* - `music`: style the speech as if talking about music, videos or other multi-media content *(only available in English (US) and not compatible with `<voice>`)* - `news`: style the speech similar to TV or radio news hosts *(requires the Matthew, Joanna or Lupe voice, only available in English (US) and English (AU))*	❌	❌	✅	❌	❌

Element	Attribute	Description	LSML (lecture.js)	SSML	Amazon Polly	Google Cloud TTS	OpenMARY
`<amazon:effect>`		applies an effect to the voice	❌	❌	✅	❌	❌
	`name`	defines the name of the specific effect to apply to the voice - `whispered`: applies a whispering effect - `soft`: speaks softly - `drc`: adds dynamic range compression	❌	❌	✅ - `soft` and `whispered` not available with Neural Voices	❌	❌
	`vocal-tract-length`	defines the tonal quality (so you can differentiate between speech even when using the same voice) - uses absolute percentage values, e.g., *110%*, or relative percentage values, e.g., *+50%* or *-20%*	❌	❌	✅ - not available with Neural Voices	❌	❌