communicated by:
Lehr- und Forschungsgebiet Informatik 9

Prof Dr. Ulrik Schroeder



The present work was submitted to Learning Technologies Research Group
Diese Arbeit wurde vorgelegt am Lehr- und Forschungsgebiet Informatik 9

A Learning Analytics Monitoring Tool for Supporting VR Developers

Ein Learning Analytics Monitoring Tool zur Unterstützung von VR Entwicklern

Bachelor-Thesis
Bachelorarbeit

presented by / von

Thiesen, Georg
433887

Prof. Dr.-Ing. Ulrik Schroeder

Prof. Dr. rer. nat. Horst Lichter

Betreut von/ Supervised by
Sergej Görzen M.Sc. RWTH

Aachen, September 19, 2024

Contents

List of Figures iii

List of Tables iv

1 Introduction 2
- 1.1 Motivation . . . 3
- 1.2 Goal . . . 3

2 Scientific Background 4
- 2.1 eXtended Reality for Education . . . 4
- 2.2 Improving learning experience with Learning Analytics . . . 4
- 2.3 xAPI: The standardization of learning data . . . 5
- 2.4 Software development . . . 6

3 Related Work 8
- 3.1 OmiLAXR Ecosystem . . . 8
  - 3.1.1 The xAPI Definitions Registry . . . 8
  - 3.1.2 xAPI4Unity . . . 9
  - 3.1.3 The Researcher Companion Panel: ReCoPA . . . 10
  - 3.1.4 OmiLAXR Framework . . . 10
- 3.2 xAPI Storage and Monitoring Tools . . . 11

4 Concept and Design 13
- 4.1 Requirements analysis . . . 13
  - 4.1.1 Participants answers . . . 13
  - 4.1.2 Discussion . . . 14
  - 4.1.3 Result . . . 15
- 4.2 Integration into the ecosystem . . . 16
- 4.3 The YetAnalytics Learning Record Store . . . 17
- 4.4 Internal architecture . . . 17
  - 4.4.1 Backend . . . 17
  - 4.4.2 Frontend . . . 18

5 Implementation 19
- 5.1 Used programming languages and frameworks . . . 19
  - 5.1.1 Developing desktop apps with Tauri . . . 19
  - 5.1.2 Efficient and memory-safe with Rust . . . 20
  - 5.1.3 The fitting frontend framework . . . 20

5.2 Implementation of LiMoxAPI 21
5.2.1 Connecting and retrieving statements 21
5.2.2 Data processing 22
5.2.3 Visualizing the information 22
5.3 The resulting application 22

6 Evaluation and Results **24**
6.1 User Evaluation 24
6.1.1 Participants answers 25
6.1.2 Discussion 26
6.2 Technical Evaluation 26
6.2.1 The setup 26
6.2.2 Performance results 27
6.2.3 Discussion 27

7 Future Work **29**
7.1 Upgrading features 29
7.2 Improving LiMoxAPIs architecture 30
7.2.1 Integrating the LRS 30
7.2.2 Modularization of features via plug-ins 30

8 Conclusion **31**

Appendix **31**

A Bibliography **32**

B Assessment Questionary **35**

Glossary **38**

Acronyms **39**

List of Figures

2.1 The Learning Analytics process cycle 5
2.2 An example xAPI statement in the JSON format. 6
2.3 The classic Waterfall Model. 7

3.1 The infrastructure of the OmiLAXR ecosystem. (Adapted from [HEGS22]) 9
3.2 Integration of xAPI in OmiLAXR. (Copied from [GHS24]) 11
3.3 Screenshot of the YetAnalytics LRS dashboard. 12

4.1 A mockup of the main dashboard of LiMoxAPI. 15
4.2 The concept architecture of the local monitoring tool. 16
4.3 The concept of the internal architecture of LiMoxAPI. 18

5.1 The resulting interface: Connection panel. 23
5.2 The resulting interface: Dashboard. 23

6.1 Processing time medians 28

List of Tables

6.1 The hardware specifications of the hosting test system 26

Abstract

The RWTH research group Learning Technologies (i9) developed an ecosystem that makes Learning Analytics (LA) available for Virtual Reality (VR) learning environments. For this purpose, the xAPI specification was used to standardize the learning data generated by these VR applications and their corresponding communication.
After its creation, the learning data is sent to an xAPI database, an Learning Record Store (LRS), from where it can be used for further usage. However, the developer of the VR learning environments have a hard time testing their application as the LRS used by the research group is not always available and does not provide the needed functionalities to check the generated learning data for correctness and completeness properly.
Therefore, this project focused on addressing this problem by answering the following questions. How can the development process of VR learning environments that use xAPI be supported? Which features should added to the Open and modular integration of Learning Analytics in XR (OmiLAXR) ecosystem to facilitate the verification of generated learning data by developers?
The monitoring tool Live Monitoring of xAPI statements (LiMoxAPI) was developed in the scope of this thesis to solve these issues.

Chapter 1 Introduction

VR is a modern and trending technology with diverse use cases and opportunities. It provides a completely new experience for video game players. In the industry, VR enables designing, developing, and evaluating digital prototypes, leading to a much cheaper and more flexible development process than with a physical construction [BV17]. And for education, it provides individual learning environments where users have lab-based experiences. An exemplary use-case is training for future doctors who can practice difficult operations under almost natural conditions [HPH+17]. But VR is also highly valuable for more theoretical topics. One example is the RePiX rendering pipeline, a project of the RWTH research group Learning Technologies (i9), to study the 3d rendering process [HGS+22].

Not to forget that the learning experience is not only of interest to the user but also to several other stakeholders. Researchers, for example, are interested in the learning process's insights. And VR gives a great opportunity here. With a digital learning environment that creates an immersive learning experience similar to a real-world equivalent, this virtual alternative enables one to monitor every event and interaction. Therefore, the technical setup is needed, meaning tools and components that enable or facilitate the creation of learning data, processing it, and making it available for the interested parties.

The research group Learning Technologies developed an ecosystem for integrating LA in VR using xAPI [HEGS22]. However, the component to store and observe the learning data, the Learning Locker or, more precisely, a LRS, is heavy and does not fit very well the needs of developers regarding testing the creation of learning data. Therefore, this project aimed to address this exact problem by providing a tool to monitor learning data that is generated in VR learning environments using the specification xAPI.

The thesis is structured as follows. After this chapter, where the thesis is introduced with its motivation and goals, the scientific background is presented with the role of xAPI facilitating Learning Analytics for VR and the background of software development. The following chapter shows related work with a focus on the research group's ecosystem: OmiLAXR. Afterward, the concept and design of the resulting tool of this project are explained, and requirements are analyzed. It follows a chapter about the more technical aspects: the implementation and used frameworks. The thesis ends with the evaluation of the tool, its discussion, future work, and the resulting conclusion.

1.1 Motivation

The OmiLAXR ecosystem provides the necessary infrastructure for the application of LA for VR application [HEGS22]. It makes VR learning environments accessible to different stakeholders. Researchers are provided with the interface of a Learning Locker to observe the resulting learning data of the learning environments, teachers can monitor their students during the learning process, and VR developers got a framework to support the integration of xAPI for VR apps [GHS24]. More technical background in chapter 2.
There were still some issues when considering the perspective of the VR developer. The Learning Locker is a web server, meaning when the developer wants to test the educational VR app, the connection to the Learning Locker needs to be established and, of course, has to be continuously reliable. These issues were appealed by developers in a study conducted in 2023 by the learning technology research group [GHS24]. Developers complained about the complexity of the infrastructure, the unreliable connections to the different components, and the time-consuming setup process. This process becomes very uncomfortable and expensive, especially when the developer wants to quickly test the VR application to determine whether the learning data is generated and processed as intended. An option would be to set up a Learning Locker locally. However, currently, it only fully supports Ubuntu and the whole configuration process is complex and requires Linux administration knowledge [Lea24a]. The issues and requirements are elaborated in more detail in 4.1. Next, the specific goal of this project is presented.

1.2 Goal

This project aimed to provide a tool that helps developers working with OmiLAXR ecosystem to test their VR application. The tool was named LiMoxAPI; this name will be used from now on for clarification. LiMoxAPI does not replace any components of the OmiLAXR ecosystem but extends it by providing an alternative to reduce the overhead of configuring connections to dashboards and learning record stores and simplifying the testing process. The requirements of LiMoxAPI were to run locally on the machine next to the VR environment, which could be an app for any of the standard operating systems (Linux, MacOS, Windows). Therefore, it has to be lightweight, cross-platform, easily deployable, and quick to set up. It should be emphasized that LiMoxAPI focuses on the needs of the VR developers and is not intended for researchers or other stakeholders who want to analyze or work with the learning data. Nevertheless, it is possible to get some insights that are of general interest, for example, the logged statements that are displayed immediately and show what the student is currently doing.

Chapter 2 Scientific Background

This chapter provides the necessary scientific and technical background information for this project. It starts with a short introduction to VR, or rather, eXtended Reality, and its relevance for education. Then, the application of learning analytics is presented, followed by an introduction to the technical standard xAPI. The last section presents the software development background used for this project.

2.1 eXtended Reality for Education

With the development of head-mounted displays (HMD), an entirely new level of immersion is achieved in VR. It is not only possible to interact with the digital world, but it also blurs borders, and the perception of being in a non-physical world vanishes [RMFW20]. This degree of disconnected awareness is known as *immersion*. When people talk about VR, they usually mean the superordinate eXtended Reality (XR), which includes not only VR but also Augmented Reality (AR) and Mixed Reality (MX). Here is a short explanation of these three terms. There are three main technologies that can help to realize such an experience. The most famous one is VR. By wearing an HMD, the user steps into a complete virtual world visually completely disconnected from his real environment. With AR, you can still see your real surroundings, but virtual objects are added to augment your interaction possibilities. Similar to AR but not necessarily strictly separating real and virtual objects is MR, where reality and virtuality are intertwined.
XR has many application areas, like in the gaming scene or the industry [CRHURCC22]. However, not only does the industry reach a new standard 4.0, but education does as well [Hus18]. Moreover, XR plays a significant role here as a technical facilitator for this new level of learning. XR motivates students due to immersive interaction and autonomous exploration opportunities [MGMADGM17]. These virtual multimodal environments provide a learning experience that is impossible with analog learning materials in the first place: Lab-based learning is a vast improvement to the learning process as users gain a much deeper understanding of their material [MM07].

2.2 Improving learning experience with Learning Analytics

Generally, from the point of view of a researcher, it is not satisfiable just to provide learning applications for students to learn with it. These learning environments need to be developed further and adapted based on feedback. The point of interest is

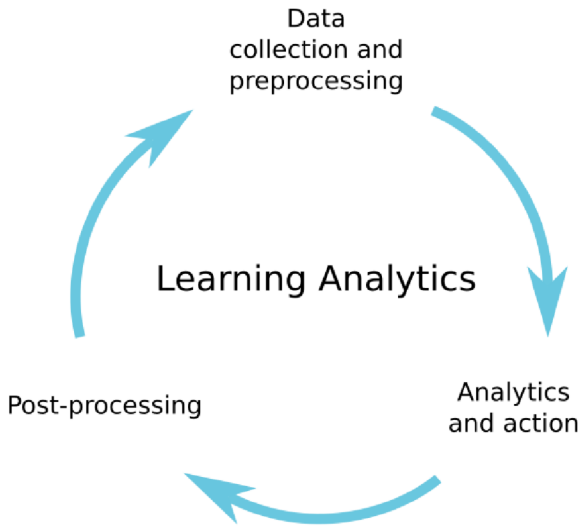


Figure 2.1: The Learning Analytics process cycle (Redrawn from [CDST12]).

the opportunity to observe and analyze the behavior of the learners and adapt it, also known as LA. The term LA is not uniformly defined but generally covers the extraction of useful information from gathered learning data as well as developing respective methods and tools to improve the learning experience as stated by Chatti et al. [CDST12].
They describe the LA process as a looping set of steps as seen in figure 2.1. The first LA step is the *data collection and pre-processing* and includes collecting learning data from different educational sources and the first processing, which includes filtering and formatting. The next step is *analytics and action*: The pre-processed data gets analyzed to retrieve patterns. With the following actions, the data gets visualized and used to predict and improve the learning experience. In the last step *post-processing*, the analysis gets adapted by modifying metrics and methods for the next iteration. This sequence restarts from the beginning and fine-tunes with each iteration.

2.3 xAPI: The standardization of learning data

Different learning contexts make comparing and exchanging learning experiences quite challenging. Before working with learning data, it has to be extracted and formatted. And what happens when this data needs to be processed with a different data set? Addressing these problems is the major goal of the eXperince API. This technical specification is to standardize the structure of the learning experience and how these learning structures can be exchanged [Ini24].
It helps to set the ground for developing apps and services that implement this standard. xAPI structures learning experience into statements. Each statement is a representation of a user (learner) interaction. These statements are tuples consisting of an actor, a verb, and an object. The actor is the experiencing learner, the verb is the action, and the object is another statement, an actor, or an activity [HEGS22]. The statement can be extended with a result and context field for more extensive usage. The result field is for a measured outcome, and the context field is for additional information [XAP24].
This vocabulary can be defined and stored in registries that are central storages,

```
{
    "actor": {
        "mbox": "mailto:jan@example.org",
        "name": "Jan"
    },
    "verb": {
        "id": "http://example.org/verb/finished",
        "display": { "en-US": "finished" }
    },
    "object": {
        "id": "http://example.org/activity/activity-1",
        "definition": {
        "name": { "en-US": "Activity 1" },
        "type": "http://example.org/activity-type/generic-activity"
        }
    }
}
```

Figure 2.2: Screenshot of an example xAPI statement in the JSON format.

allowing other users to use previously created definitions. A registry is provided for the VR ecosystem, where the definitions can be stored in JSON format. Using JSON is a trade-off between human and computer-efficient readability. When working with xAPI, a database is also needed to store and request the learning data. xAPI defines this kind of database under the name Learning Record Store as "a server (i.e., a system capable of receiving and processing web requests) that is responsible for receiving, storing, and providing access to Learning Records." [Sof24].

2.4 Software development

Developing software is not a spontaneous act of writing some lines of code. It is highly important to structure the development process. However, there is not one preferred way of realizing a software product. There are many different models for software engineering, two examples are the Waterfall and the Spiral model [Pre05]. These models are known as software development life cycles that repeat over and over again to improve the software further. As this bachelor's thesis is time-limited, only the first iteration is realizable. However, the Waterfall Model (Fig. 2.3) was a supporting guideline for the development process of LiMoxAPI.
The following steps of the Waterfall model are described in [Ude21]. Analyzing the requirements is the first phase, which includes identifying the customer's needs and filtering the necessary features to implement for the following design phase. In this phase, the architecture of the resulting software is planned, as well as the design patterns to use and the concept of the graphical user interface. In the implementation phase, the planned concepts and designs are turned into the resulting software by programming the application. The following testing phase includes performance tests as well as overall operability checks and generally if the specified requirements are satisfied. Naturally, the last phase of the Waterfall model is maintenance, which

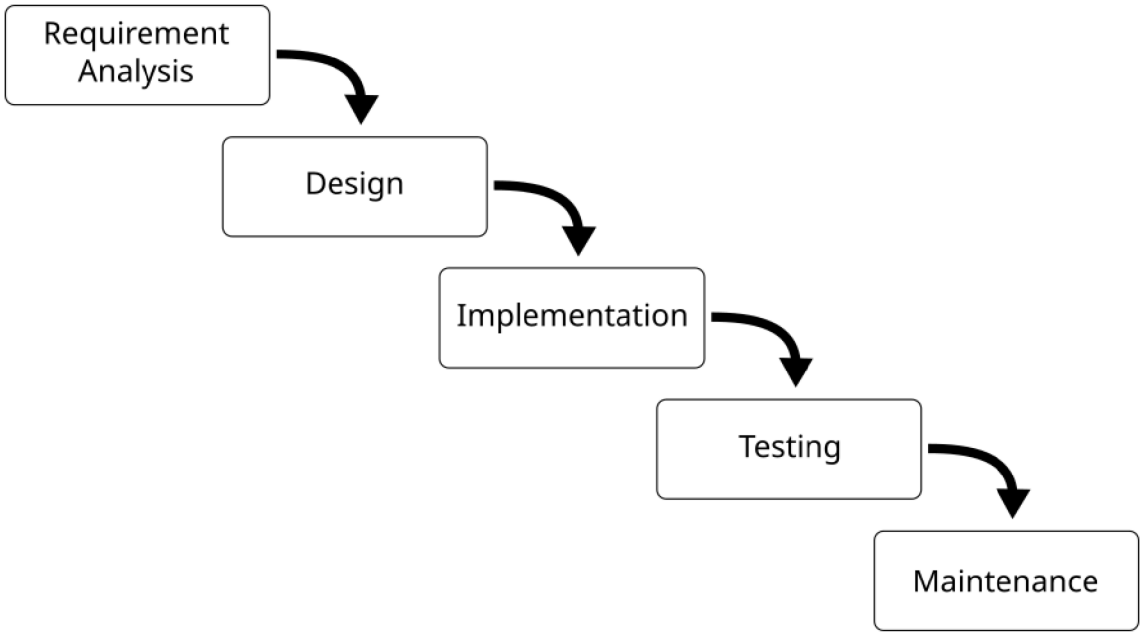


Figure 2.3: The classic Waterfall Model. (Adapted from [Ude21])

involves solving emerging problems with the shipped software. Except for the maintenance phase, which is not realizable in the scope of this project, these steps were the guideline for the development process of LiMoxAPI.

Chapter 3 Related Work

LiMoxAPI, the tool that was the result of this project, is embedded in a more extensive project of the research group Learning Technologies, the OmiLAXR ecosystem [HEGS22]. That ecosystem aims (Fig. 3.1) to provide the necessary infrastructure for different stakeholders for integrating LA through xAPI into VR environments. These stakeholders have different profits from this ecosystem. Researchers, for example, need to record and analyze the generated learning data of students or teachers who want to observe the student's learning process. And not to forget, are the developers of the VR learning environments.
The implementation of LiMoxAPI focused on the developer's needs, as it is quite challenging to monitor the in-process VR app for test purposes. Therefore, LiMoxAPI is used as an extension to the ecosystem that monitors VR apps and observes if learning data is generated as intended.
This chapter will present the OmiLAXR ecosystem, which is the foundation for LiMoxAPI, and further tools that store or monitor xAPI statements.

3.1 OmiLAXR Ecosystem

The OmiLAXR ecosystem is an infrastructure consisting of several components to standardize the usage of xAPI for educational VR apps. This ecosystem is intended for a diverse group of stakeholders: Researchers who need an interface to retrieve the learning data for scientific purposes. Teachers and coaches get the possibility to observe what their students are doing and to evaluate learning processes. But not to forget are developers who need a unified system to develop VR apps. The overall architecture can be seen in Figure 3.1 In the following are the main components of the OmiLAXR ecosystem presented.

3.1.1 The xAPI Definitions Registry

When working with xAPI, the activities, verbs, and actors have to be defined with an identifier. As the xAPI standard intends, this identifier should be a URL so that the definition is uniquely identifiable and can be used by others. These catalogs of definitions are specified by xAPI as *Registries* [XAP24]. For the OmiLAXR ecosystem, a registry is set up, specialized on VR definitions. Every xAPI definition is stored in a registry in the JSON format. With the help of git, versionization of the registry and referencing former definitions are enabled.
Usually, JSON is a good balance between human and computer readability but is

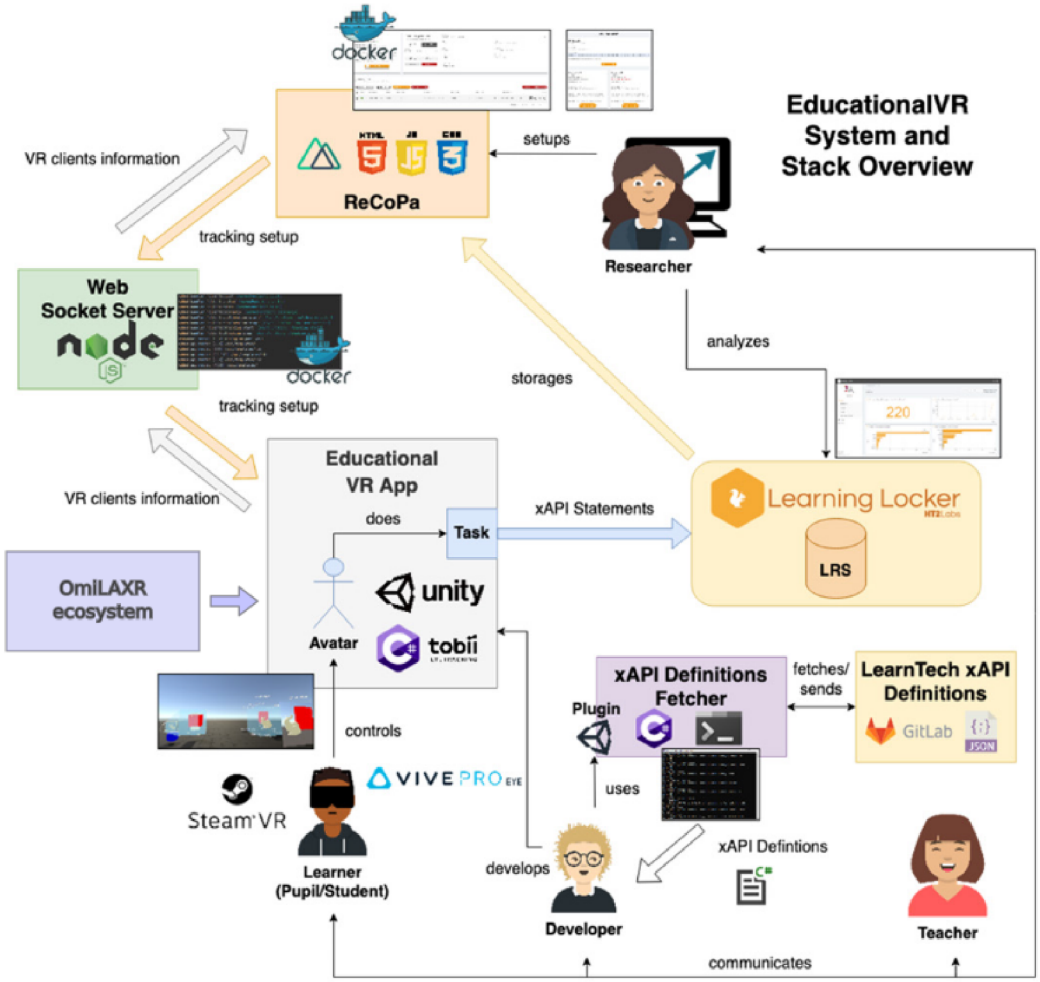


Figure 3.1: The infrastructure of the OmiLAXR ecosystem. (Adapted from [HEGS22])

unsuitable for recurring definition interaction. Also, the usage of git for non-it enthusiasts can get quite challenging. Therefore, a registry frontend is provided in the form of a website to access a wide range of approved definitions, with the possibility of suggesting new ones. The research group collected over 500 definitions as a starting vocabulary to use [HEGS22].

3.1.2 xAPI4Unity

With the xAPI Definitions Registry, developers have the possibility to access proven definitions for generating xAPI statements. Nevertheless, the creation of these statements is quite cumbersome. The definitions still have to be copied into C# code, an awkward, repetitive work that is highly error-prone. This comes with the problem of synchronization. When the registry gets updated, all the hard-coded statements must also be manually updated.
The Unity plugin xAPI4Unity[1] (formerly known as the xAPI Definitions Fetcher) addresses this problem by generating C# classes out of the defined definitions. Verbs

[1] https://gitlab.com/learntech-rwth/omilaxr-ecosystem/xapi-4-unity, last accessed: September 2024

and activities are members of the enveloping definition, and methods provide the extensions. When new definitions are created, the plugin transforms them automatically in the C# classes, and with the support of IntelliSense, code completion is enabled as well [HEGS22].

3.1.3 The Researcher Companion Panel: ReCoPA

The xAPI Definitions Registry and xAPI4Unity focused on the setup in the background to enable standardized LA with xAPI. But to work with the generated data, which is the actual point of interest for the researcher, a user interface is needed, and that is provided by the Researcher Companion Panel (ReCoPA). It includes the following components. A connection interface to connect to a specified VR learning environment where filters can be applied to filter for specific gestures and objects. The Panel provides also a monitoring tool that captures all incoming statements and reduces them to the most relevant information.
The incoming statements are not directly sent from the VR app but from the LRS to which the VR app is connected. Due to the user-friendly UI, it is not limited to experienced developers but is also easily accessible to non-computer scientists like researchers or teachers. ReCoPA has some similar functionalities as LiMoxAPI but with a different use case. Instead of making learning data accessible to different stakeholders, LiMoxAPI focuses on the developer to observe if the learning data is generated correctly. Nevertheless, some of ReCoPA's functions were an inspiration for LiMoxAPI.

3.1.4 OmiLAXR Framework

The following details are referenced from [GHS24]. During the development of VR learning applications like RePiX VR [HGS23], the research group extended the xAPI Definitions Registry with definitions especially VR related ones like eye tracking or head movements to improve the insight of what happens inside the learning environment. This resulted in a whole framework to support the integration of LA in XR applications: The OmiLAXR framework[2]. Its purpose is to standardize and facilitate the integration of LA into VR applications [GHS24]. The approach can be seen in Figure 3.2.
To support third-party libraries that, for example, provide sensors for eye tracking, OmiLAXR works with adapters that function by the "Plug & Play" principle. These adapters match the libraries to the learning context used by OmiLAXR. This learning context is represented by a Learner, a Learning Scenario, and Virtual Objects. These entities are the basis of every VR learning environment. There is currently a new approach in development, though. Another component of OmiLAXR is the Main Tracking System that facilitates different sub-tracking systems and generates the resulting xAPI statement. Each statement includes the following information: The XR application that created the statement, the version of the app, a possible game mode, and the current task context. The statements are then send to the configured LRS or cached otherwise.
However, OmiLAXR lacks a supportitive testing tool. How do you know that the

[2] https://docs.omilaxr.dev/, last accessed: September 2024

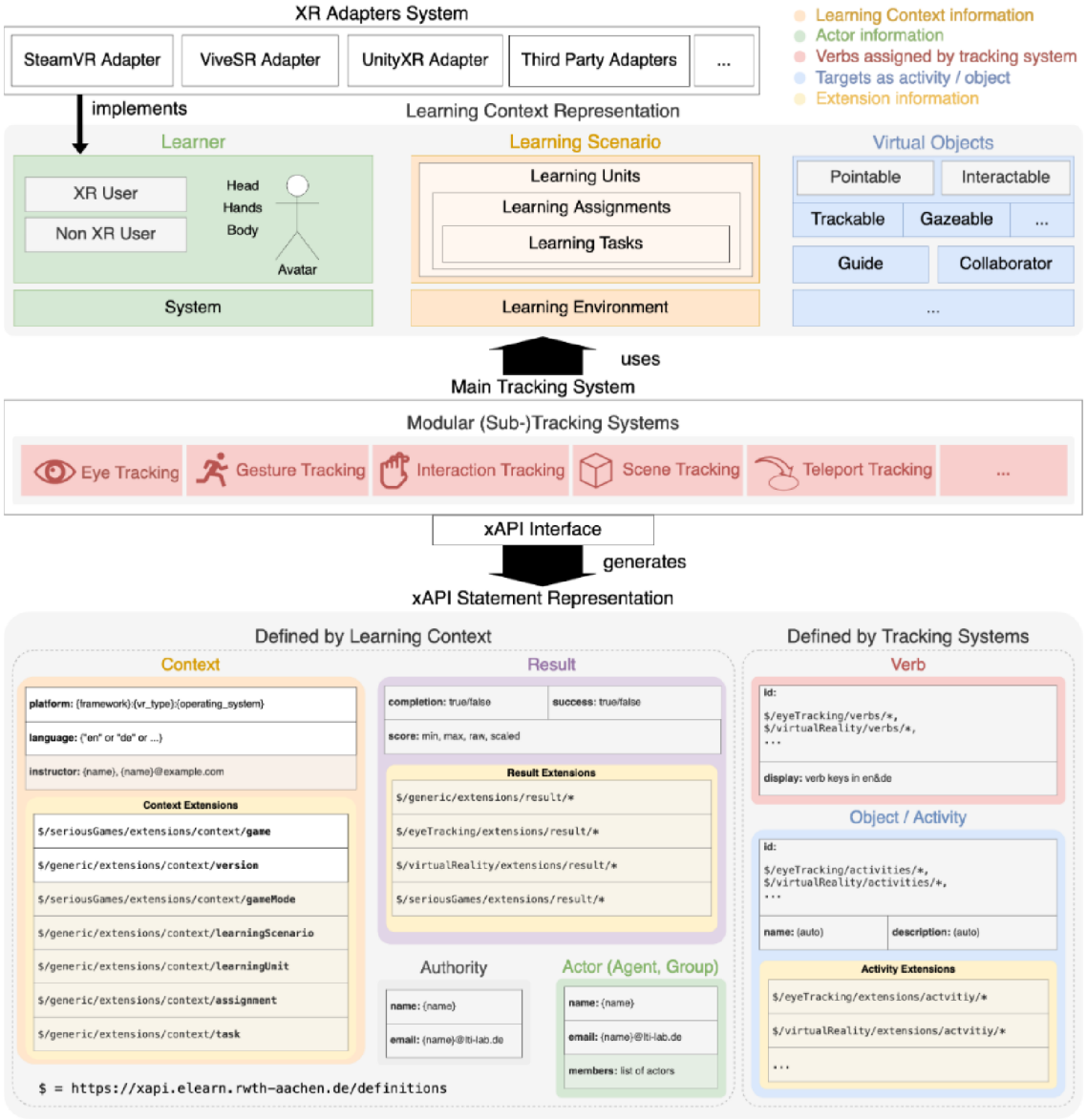


Figure 3.2: Integration of xAPI in OmiLAXR. (Copied from [GHS24])

learning data generated with OmiLAXR was created correctly and sent successfully to the LRS? This is where LiMoxAPI comes into play to fill that gap.

3.2 xAPI Storage and Monitoring Tools

When working with xAPI, needed as well is a LRS: A database with a web service not only for receiving and processing requests but optional equipped with LA tools as well as a monitoring dashboard. An open-source variant of an LRS is the Learning Locker[3] that is also used by the research group. Learning Locker utilizes MongoDB as an underlying database. It is highly customizable and offers the possibility to create versatile visualizations. However, this leads to a more complicated setup process.

[3] https://learninglocker.atlassian.net/wiki/spaces/DOCS/overview, last accessed: September 2024

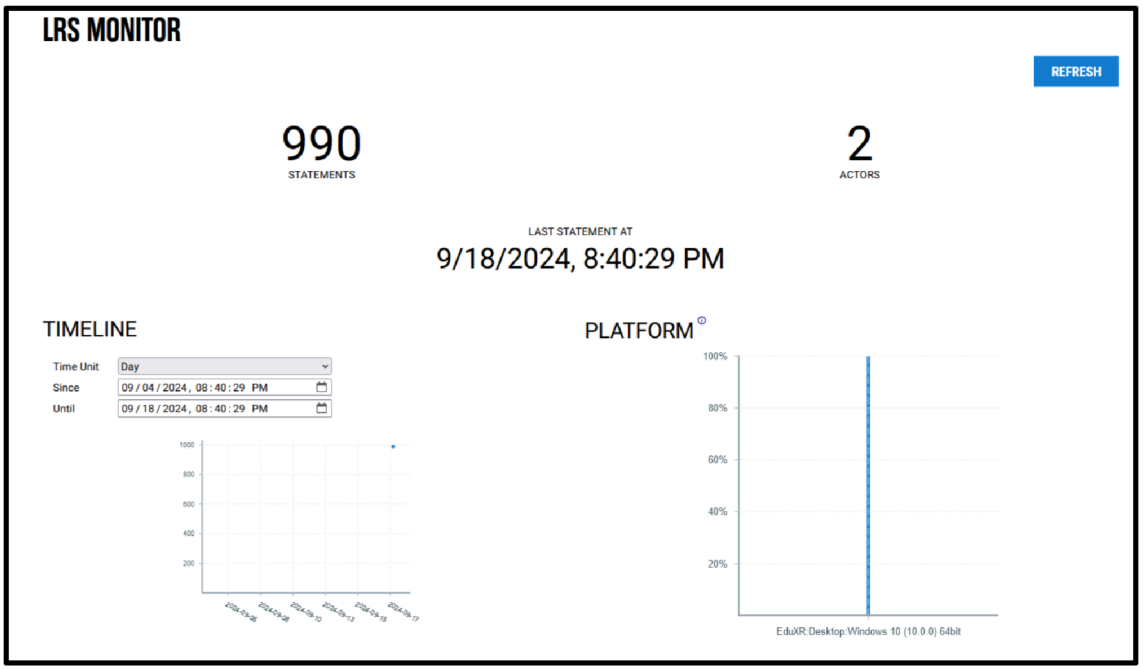


Figure 3.3: Screenshot of the YetAnalytics LRS dashboard.

Another option is the YetAnalytics LRS[4], which uses SQL database management systems. Deploying the YetAnalytics LRS is relatively easy, as most of the configurations are already done. It has a simple interface to set up credentials and the admin account and provides statement browsing and a simple monitoring dashboard (Fig. 3.3). Nevertheless, the dashboard is limited and lacks features like filtering statements, which is a functionality in demand for testing xAPI VR environments, as shown in chapter 4.1. The fully configured YetAnalytics LRS can be zipped and deployed on another system, which simplifies the usage even more.

[4] https://www.yetanalytics.com/, last accessed: September 2024

Chapter 4 Concept and Design

After the introduction to related work, this chapter presents the technical concepts of LiMoxAPI. It begins with the requirement analysis in the form of a discussion round. It follows an overview of the integration of LiMoxAPI into the OmiLAXR ecosystem. The last section presents the inner architecture of the monitoring tool.

4.1 Requirements analysis

The main issues when working with the OmiLAXR ecosystem were already displayed in the user study done by the learning research group regarding the OmiLAXR framework[GHS24]. Developers complained about connection issues when using the Learning Locker of the research group, as well as the time-consuming setup. However, the focus of this study was specifically the usability of OmiLAXR. Therefore, a requirement analysis was conducted to pin down the challenges of developing educational VR environments and to work out solutions on how to facilitate the development. Finding developers who use the OmiLAXR framework to build VR applications was quite challenging. Nevertheless, two developers who had experience working with xAPI and especially working with OmiLAXR were found and willing to participate in this requirement analysis.
First, the background and motivation for implementing the tool LiMoxAPI were presented. Then, the participants were given a sheet of questions to fill out, and afterward, the comments were discussed together. The assessment was based on the following questions with the workflow in the given order:

1. How do you generate xAPI Statements?
2. How do you verify the correctness and completeness of the generated data?
3. How do you visualize the data during the development process?
4. Which functions shall be provided to optimize the development process?

4.1.1 Participants answers

In the following subsections, the stated answers are presented according to the lead questions.

Generating Data

Both developers stated to create xAPI statements via OmiLAXR and the provided components, such as the eye-tracking component. The problem, however, was to check if the statements were generated successfully. The mentioned method was to check the LRS manually if the statements were created. The same problem occurred for verifying correctness and completeness.

Verification of Data

Again, verifying the correctness and completeness of the data was more or less challenging. The current procedure was to check the LRS for new statements. Prepared scenarios where specific cases to generate data were tested were mentioned as a way to verify whether the statements were generated according to the scenario, but still, a follow-up check-up in the LRS was necessary. However, the real problem with manually checking was that it is quite time-consuming to trigger sending a specific statement and check the LRS if it arrived there, which prefers a second person observing the LRS to notify if the intended statements have arrived.

Visualization of Data

The visualization of data was stated as difficult, at least to generalize, as every project needs individual tables and plotted information. As suggested, showing the numbers of incoming statements is useful in general.

Optimizing functions for the development process

Both developers stated that classifying or filtering statements is highly valuable. Filtering statements was already possible with the dashboard ReCoPa, but its functions are limited. It is not possible to filter for specific extensions, for example.

4.1.2 Discussion

The assessment showed that the major problem was verifying whether the learning data was correctly created or even generated at all. Checking up a LRS is time-consuming and sometimes not even possible due to connection issues. An important problem to address was triggering a statement by performing a special action in the VR environment with the head-mounted display on and parallel checking the LRS if the statement arrived. This feat is impossible unless you have a partner. Another issue to solve was confirming the creation of specific statements, as it is quite challenging to make out a specific statement in an LRS when several statements arrive each second. The assessment confirmed the need for a tool as planned for LiMoxAPI: A monitoring tool that runs locally next to the VR app and observes the creation of xAPI statements. The discussed solution for checking for specific statements was to use a filter system that allows filter-in and filter-out statements accompanied by a triggered notification sound whenever that filtered statement emerges. These audible notifications solve the problem of parallelly checking statements in the LRS while testing the VR app. Another point of the discussion was the method of monitoring. The suggestion to monitor the statements directly in the VR app and not the statements incoming in the LRS came up

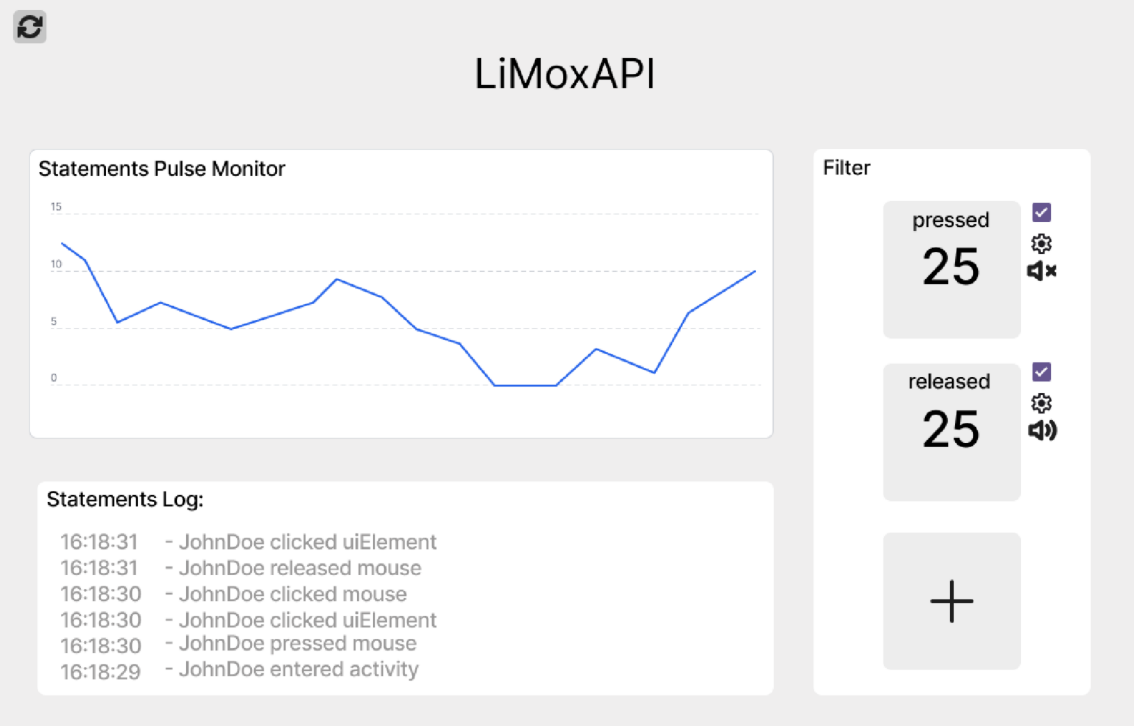


Figure 4.1: A mockup of the main dashboard of LiMoxAPI.

in the discussion, which may seem reasonable at first. The issue here is how to monitor the statements directly exactly. One solution is to send the statements to LiMoxAPI, but that means it has to include a LRS, and that is not really different from setting up a local LRS next to LiMoxAPI and the VR app. Using an external LRS also has the benefit of being modular. The only setting that has to be changed is the connections credentials when switching between LRSs. Another benefit of monitoring the LRS and not the VR app directly is that it also tests if the statements are successfully sent to the LRS. However, the solution to extend LiMoxAPI by an LRS is a viable option and is elaborated in more detail in the chapter Future Work (7).

4.1.3 Result

The discussion round, in which the requirements were assessed, resulted in an overall concept of LiMoxAPI with the following main features. To evaluate how many statements are incoming, a *pulse monitor* is necessary that shows live the number of new statements arriving like a heartbeat monitor. The *logging console* is to get more information about the statements by logging all incoming statements and displaying the timestamp of creation, the actor name, the verb, and the object, each without the whole URL identifier. The main feature is the *filtered counter* that lets the user create multiple filters to count the statements as specified in the individual filter options. Important for the filter is that is able to filter for every nested value in the statement. Therefore, to filter for a specific value that is somewhere deep nested in the statements tree, the according path to that value has to be specified. As an example: Every xAPI has an actor and every actor has a name. When filtering all the statements that have an actor with the name "Player1," the path "actor.name" has to be specified as well

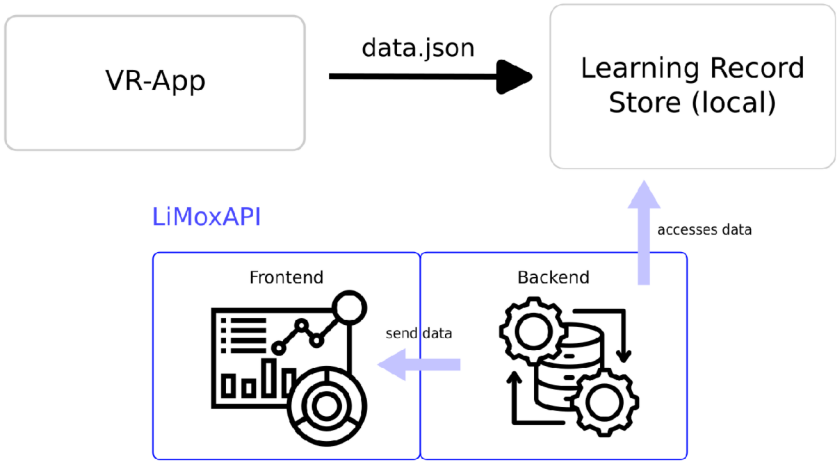


Figure 4.2: The concept architecture of the local monitoring tool[1].

as the value "Player1". This enables to filter flexibly for any desired value. Important as well is the functionality to filter out statements or, more precisely, to count every statement that does not include the specified value. As discussed with the developers in the evaluation, a filter is an important functionality to test the generation of specific statements, such as eye movements. One problem was still the complicated process of checking up if the targeted statements were created as intended. One emerging solution was to integrate feedback for incoming statements in the form of audible notifications.
A first mockup (Fig. 4.1) with the described features was designed as a plan for the following implementation process. The upper left component is the pulse monitor. It displays the amount of new, freshly incoming statements. In the bottom left corner is the logger console. The logger shows every incoming statement in an easy-to-read format. The component on the right side is the filtered counter. These features were the base functionalities of LiMoxAPI.

4.2 Integration into the ecosystem

After assessing the requirements and designing a first mockup, an architectural concept of the integration of LiMoxAPI into the OmiLAXR ecosystem had to be developed. The first aspect to cover is modularity. As already mentioned, LiMoxAPI should not replace anything but extend it; therefore, instead of monitoring the VR application directly, it has to monitor the LRS to which the VR application is connected. That allows a simple integration of LiMoxAPI because developers can deploy their VR app and connect it to the chosen LRS as before, just with the addition of LiMoxAPI to start and connect to the LRS. An overview can be seen in Figure 4.2. It is quite similar to the ecosystem but broken down into the necessary components for the requirements. The VR app is still sending the xAPI statements to an LRS. That enables modular usage. However, for the usage of LiMoxAPI, it is suggested to use a local LRS, which

[1] This figure has been designed using resources from Flaticon.com

is running next to the VR app, and the monitoring tool on the same device to establish a fast and reliable connection. It is important to remember that the stored statements are primarily for the monitoring process, not for further processing. Nevertheless, the integration of an LRS to this monitoring tool could be a valuable extension for the future, but more on that in Chapter 7. In the next section, a recommended LRS is presented as a local deployable alternative.

4.3 The YetAnalytics Learning Record Store

In order to work fast, reliably, and without a complex setup, LiMoxAPI needs an LRS with the same attributes. As mentioned, the Learning Locker, which is deployed by the research group, is not suitable for this task as it is not always available and is not as fast as a local variant. An option would be to install a Learning Locker locally, but this is complex and requires a Linux distribution first and, secondly, further knowledge of Linux administration, which is not within reason. However, another option is the Yet Analytics SQL LRS[2]. It works on all three major operating systems, Windows, Linux, and macOS, and is easy to install. For LiMoxAPI, this LRS is even a recommended solution because it is provided as a fully preconfigured compressed package. The only requirement is a Java Development Kit on the system. The preconfigured settings include credentials for communicating with the REST-API of the LRS that automates the connection process of LiMoxAPI to the LRS. Also preconfigured are the credentials for the web panel that provides some simple insights into the database and an interface to manage the API credentials.

4.4 Internal architecture

Not only does the usage of LiMoxAPI have to be modular, but also the inner functionalities. In this case, these modular components are requesting, processing, and visualizing the learning data. This leads to a separation of the monitoring tool into two parts: a backend and a frontend. The backend fetches new data from the LRS periodically and then processes the data. Afterward, the backend sends the processed and filtered data to the backend, where the arriving information is visualized. This setup leads to a clear functional division and keeps the heavier computations and background work to the capable resources in the backend. Communication between both application layers is established via an event system. The channel allows sending payloads in both directions as it is necessary for the frontend to send a message to the backend as well, for example, the connection credentials for the LRS that are typed in by the user.

4.4.1 Backend

The main backend function is the loop with the first instruction to fetch all new xAPI statements from the specified LRS. The loop gets initiated as soon as the connection credentials are set and the connection is successfully established. Afterward, it runs until the program terminates. Formatting the statements is the next step after requesting them. The flow can be seen in Figure 4.3 on the left side. After this step,

[2] https://github.com/yetanalytics/lrsql

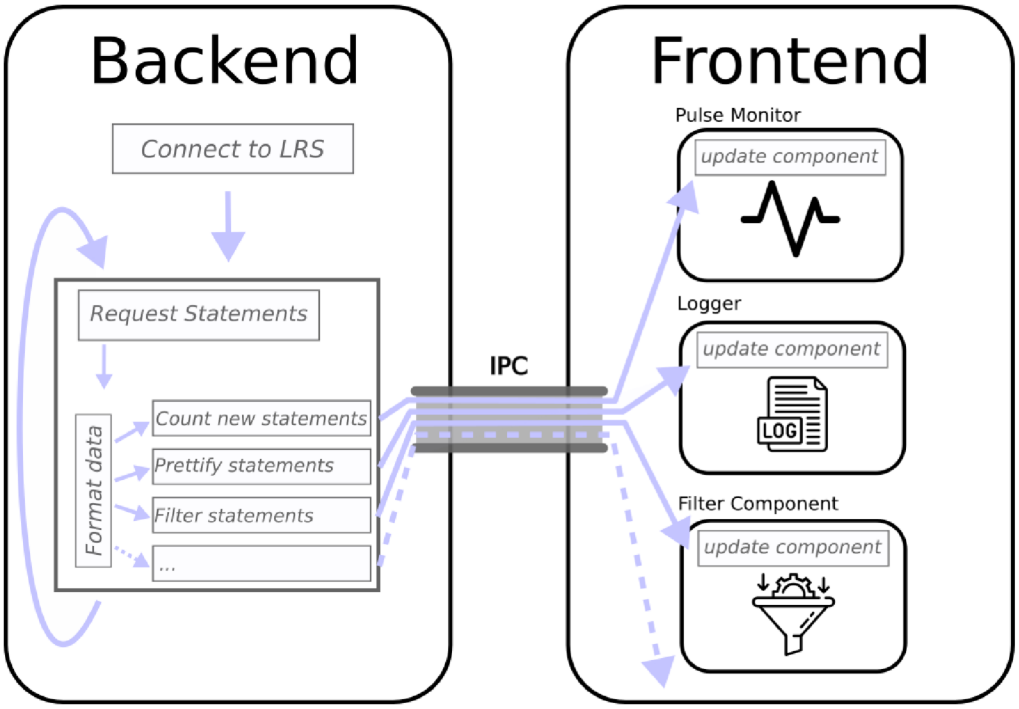


Figure 4.3: The concept of the internal architecture of LiMoxAPI[3].

the formatted data can be used for different features like the pulse monitor, the logger, or the filter. This data flow principle allows a flexible way of adding, deleting, and modifying features [Sou12]. The downside is a lack of performance. The more efficient principle would be the monolithic approach [Lam21]: Taking every feature into account and processing the data once with several outputs leads to less redundant operations and, therefore, to faster and more efficient computation. However, that results in increased complexity of updating the components, and as shown in the technical evaluation 6.2, this is not necessary with the current performance. Nevertheless, implementing these features has to be efficient, and that urges for a performant programming language for the backend.

4.4.2 Frontend

The purpose of the frontend is to visualize the data that is requested and processed by the backend. Therefore, the frontend divides the different features into individual components, as shown in Figure 4.3 on the right side. Each component is responsible for receiving the necessary data and visualizing it accordingly. The filter component consists of subcomponents that each represent a configured filter. These subcomponents are created dynamically. Each component handles the incoming data differently. The data for the filtered counter, for example, is replacing each filter value. However, for the pulse monitor, the data is added to the existing data stored in the frontend. This allows a redundant and fast update process. The front end listens to new incoming data and triggers an update function when the data arrives, which enables synchronization between the backend and the frontend.

[3] This figure has been designed using resources from Flaticon.com

Chapter 5 Implementation

As discussed in the previous chapter, LiMoxAPI has specific conditions to satisfy. It has to be fast, easily deployable, quick to set up, and cross-platform. The presented concept serves some of these requirements, but the choice of the programming language and frameworks and how they are applied are the decisive factors. In this section, the programming languages and frameworks that have been used to develop LiMoxAPI are presented, and afterward, how the individual functionalities are implemented. It starts with the presentation of Tauri, the overarching framework that enables the development of cross-platform desktop applications. This is followed by a section about Rust, the machine-close language used for the backend of Tauri apps. Afterward, the tools for the frontend are presented with TypeScript as the programming language, using Vue.js to build the graphical user interface.

5.1 Used programming languages and frameworks

After developing the overview of the application, the question emerges as to which programming language(s) to use with which frameworks. One major requirement for LiMoxAPI is to process data fast. When monitoring OmiLAXR apps, a high traffic of incoming data can be expected. That urges for an efficient programming language. However, choosing a fitting frontend framework is important as well. Each framework causes a different memory usage, bundle size, and performance. Also to consider is the maturity of the framework ecosystem. Normally, the choice of programming language is the first to make, but in this case, it goes hand in hand or even secondly with the choice of the overarching framework to build desktop apps. Therefore, this section starts with the introduction of the framework Tauri before the languages and tools for the backend and the frontend are presented.

5.1.1 Developing desktop apps with Tauri

One major challenge when developing desktop apps is building the application for multiple platforms, and one option is to use web apps that run on any device with a web browser [BL21]. Nevertheless, to build a performant application and not a full web server, another tool like the electron framework is needed [Fou24] or the more efficient alternative Tauri [Nah22]. The following technical details are referenced from the Tauri 2.0 Documentation [con24]. Instead of using a shipped-in browser engine, Tauri facilitates the systems WebView by using the WebView rendering library WRY[1]

[1] https://github.com/tauri-apps/wry, last accessed: September 2024

and TAO[2], a window creation library. Both libraries support major desktop platforms like Windows, macOS, and Linux.
The internal functionality of Tauri apps is divided into a backend and a frontend, similar to native web apps. The backend is running in Rust, which uses the Tauri ecosystem of libraries to provide the bundler, runtime to run the WebView, and further libraries to enable Tauri apps. The frontend can be built with plain HTML, CSS, and JavaScript but naturally allows the usage of any UI framework like Angular, React, or Vue.
Tauri uses asynchronous message passing for the Inter-Process Communication (IPC) between the frontend and the backend. More precisely, Tauri uses two different methods: Events and Commands. Both sides can send events that are one-way messages without a response. On the other hand, Commands allow the frontend to call Rust functions in the backend and, if it exist, receive the returned value. The frontend sends a request that triggers an invoke handler to call the actual function. The request and the response are serialized to JSON to allow calling functions of another programming language. Nevertheless, Commands can only be used by the frontend; the backend cannot call frontend functions.

5.1.2 Efficient and memory-safe with Rust

As a machine-close programming language, Rust has the perfect conditions for LiMoxAPI [3]. Its ownership lifetime management results in safer application than C or C++ regarding memory safety [FCV+21]. In addition, it has a modern and practical ecosystem in the form of easily manageable libraries (crates) and a user-friendly compiler. Another argument for Rust is the framework Tauri, which enables the development of cross-platform applications with Rust, as described above.

5.1.3 The fitting frontend framework

It is not only the backend that has to be performant. The frontend does, too, when receiving multiple times per second a stack of new statements that have to be visualized immediately. The most used JavaScript frameworks to build web app interfaces are React, Angular, and Vue.js, as shown in the StackOverflow Developer Survey 2024 [Ove24]. All three tools are modern and extensive frameworks with thriving communities. Nevertheless, native Vue.js outmatches the other two frameworks when it comes to content modification [Jav24][Sak19]. In addition, Vue seems to be more user-friendly and to has an easier learning curve, which is not to be neglected [Vya22][Sak19]. Another trending framework is Svelte. As it compiles into vanilla JavaScript code, the performance is even better than Angular, Vue.js, and React [Lea24b][Jav24]. However, it is a relatively new and upcoming framework with a less mature ecosystem. Overall, Vue.js is a good balance between performance, usability, and maturity, which is why Vue.js is the framework used to build the interface of LiMoxAPI.
Vue.js is structured in components that can even be single-file-based, meaning the JavaScript code, the HTML code, and the styling can be defined in one class, which makes the project structure organized and neat. This component-based approach is

[2] https://docs.rs/tao/latest/tao/, last accessed: September 2024
[3] https://www.rust-lang.org/, last accessed: September 2024

perfectly suitable for the concept of LiMoxAPI as the features can be separated into individual components with, if necessary, subcomponents. The framework ApexCharts.js[4] is used to plot the number of new statements for the pulse monitor.

5.2 Implementation of LiMoxAPI

The core code structure of LiMoxAPI provides the data flow infrastructure. This starts with the connection functionalities. Then, the endless loop starts, requesting new statements and formatting them in the following step. Afterward, the data gets processed for the individual features. Basic payloads for the needed data types are provided to send the data to the frontend where it finally gets visualized. This section describes how these steps are implemented with regard to the features of LiMoxAPI.

5.2.1 Connecting and retrieving statements

Under the hood, no lasting connection gets established. The xAPI statements are retrieved via HTTP requests[5]. The xAPI specification helps here with a standardized request header. Before starting the request routine, it is essential to check whether the credentials are correct and the LRS is available. Therefore, a simple GET request is packed with the provided connection parameters consisting of the URL, port, API key, and key secret. If the request is successful, the connection is labeled as *Connected*, and the following retrieval of statements starts. If a request is unsuccessful, the connection is labeled as *Disconnected*, and LiMoxAPI keeps trying to request new statements. At any time, the connection parameters can be changed.
On the one hand, LiMoxAPI has to be flexible to which LRS connect, but on the other hand, the connection process should be easy and quick for the user. Therefore, an SQL LRS is preconfigured with a config file that contains the login credentials to the web interface but also with an existing database that contains request credentials that are needed by LiMoxAPI to connect with the LRS. The whole setup SQL LRS is zipped to be shippable. The configured request credentials are preconfigured in LiMoxAPI as retrieving them by reading the database files is impossible. The connection is not automatically established with these credentials but is already typed in the connection interface to facilitate the connection process.
The request for new statements takes place in an endless loop with a delay of 300 milliseconds to not overburden LiMoxAPI with too many unnecessary requests. xAPI provides an endpoint to request statements since a given timestamp. Therefore, with the start of LiMoxAPI, the timestamp is set to the current date and time. After each iteration of requesting statements, this timestamp gets updated with the date and time when the last received statement was stored. The statements are stored and transmitted in the JSON format that allows to format the data with the serde_json create[6] into data objects.

[4] https://apexcharts.com/
[5] https://www.w3.org/Protocols/HTTP/Request.html, last accessed September 2024
[6] https://docs.rs/serde_json/latest/serde_json/, last accessed: September 2024

5.2.2 Data processing

The first step is to format the serialized JSON statements into data objects to make the contained values accessible. The serde_json crate provides the enum type *Value* that can be any of the standard data types. In this case, it is an array of further *Values*. That is how the whole JSON tree is transformed into a nested data structure. This array is then accessible for the different features by the different functions. For the pulse monitor, for example, the data array only has to be counted before the resulting value is sent to the frontend. The process for the logging feature is more complex. First, the timestamp, actor, verb, and object must be parsed from every statement. The four retrieved values of every statement then get prettified into human-readable strings and are sent to the frontend. The resulting information is sent via the Tauri event system to the frontend. Therefore, global payloads are defined to exchange different kinds of data types.

5.2.3 Visualizing the information

The data is incoming via the tauri event IPC. The visualization process is only triggered when new statements arrive, leading to a synchronization between the frontend and the backend. For this task a listener is set that reacts only to the specified events. When the package arrives the data is added to the existing data in the case of the pulse monitor and the logger console. The framework ApexCharts.js plotts the number of new statements in the pulse monitor graph. The filtered counter feature only overwrites the former values. The data values of the components are wrapped in Vues reactive objects that automatically update the context where the variable is used whenever its value changes [Vue24]. This reactivity functionality is especially useful for updating the pulse monitor realized with ApexCharts, as the graph immediately detects the changes and refreshes the visuals.

5.3 The resulting application

The resulting connection interface can be seen in Figure 5.1. When LiMoxAPI is started, the connection fields are automatically filled with the parameters and credentials specified in the configured SQL LRS. Now, the user only has to press the connect button, and if the connection process is successful, the view automatically changes to the dashboard. Of course, the login form can be filled with different values to connect to an alternative LRS. Figure 5.2 shows a screenshot of the actual monitoring dashboard. During the development process, a couple of smaller features were added. An info component between the pulse monitor and the filtered counter component shows general information: the timestamp of the last arrived statement, the overall number of monitored statements, and the number of different actors. In addition, when a statement in the logging console is clicked, a window pops up showing the full statement. Nevertheless, LiMoxAPI has room for improvement, ranging from performance aspects to whole new features, as discussed in the next section.

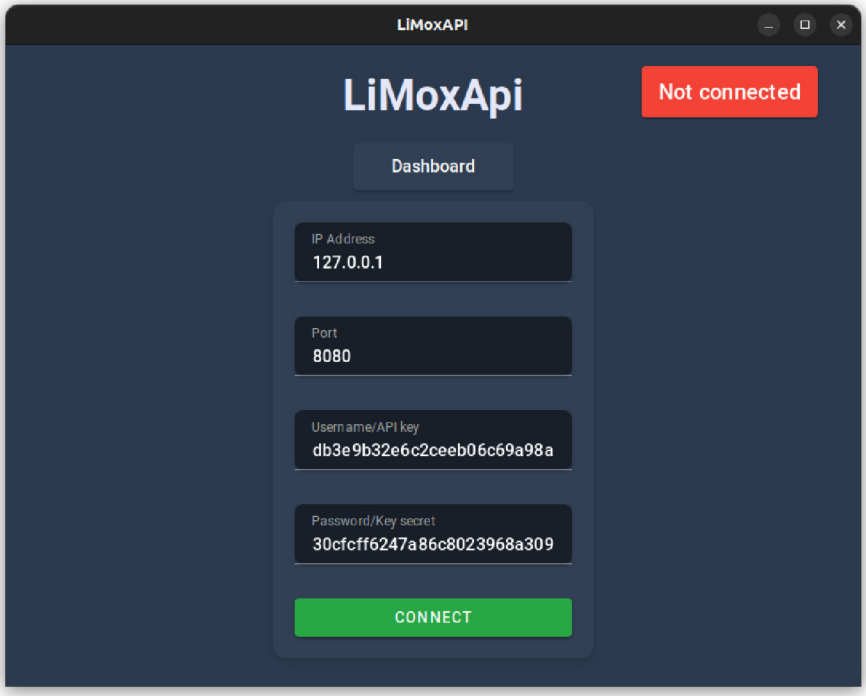


Figure 5.1: The resulting interface: Connection panel.



Figure 5.2: The resulting interface: Dashboard.

Chapter 6 Evaluation and Results

There is no dominant approach to evaluating the quality of software, as it is pretty challenging to find a model that is always fitting for every case. So adapting general methods to the specific product is necessary [MMR14]. Rounding up software development through evaluation is an essential part of the process. For LiMoxAPI, two types of evaluations were conducted. First, a user evaluation was conducted, where the developers from the requirement analysis (4.1) were invited for a second time, and secondly, a technical evaluation that revealed how robust LiMoxAPI is. So this chapter starts with the user evaluation, where the question workflow is presented first, followed by feedback from the developers. The second section presents the technical evaluation, starting with observed and measured performance and ends with an evaluation of the gathered evidence. Aftward in the last section, both evaluations are discussed.

6.1 User Evaluation

At the beginning of the development of LiMoxAPI, a requirement analysis was conducted to extract the necessary features to support developers of VR environments using xAPI; for this assessment, two developers were found. The same two participants were invited for a second round to see if the expectations and the issues stated in the requirement analysis could be met and, in general, if LiMoxAPI is a supporting addition to the OmiLAXR ecosystem. This user evaluation is described in this section. The assessment started with a live demonstration of LiMoxAPI, including the setup process and the presentation of the different features. During the procedure, both developers could ask questions or request specific actions to do. After the demonstration, the participants were given a sheet of questions to fill out. The questions in the assessment were orientated on the software quality standards of the ISO norm 25010 [ISO11] to get an extensive impression of LiMoxAPI. The subjects of the questions were as follows:

1. Extent of the functionalities provided
2. Impression regarding reliability and performance
3. Usage of the interface

6.1.1 Participants answers

After the demonstration, the participants asked questions, and the functionalities were discussed before everything was written down on a sheet of questions regarding the quality of LiMoxAPI. The feedback given to the listed subjects above is presented in the following subsections.

Functionality

Under the subject *Extent of the functionalities provided*, the developer stated the functionality to monitor incoming learning data live is highly valuable because old statements do not interfere with the current data. In general, the filter was much appreciated, especially with the sound notifications, which enable to check if the statements arrived without taking off the VR headset. However, typing in the JSON path to filter for a specific value is not intuitive, as the user has to remember the whole statement structure with the possible paths. Two recommended solutions were first to add a drop-down menu where the user types in the path that suggests the standard paths of statements that are created with OmiLAXR and secondly to enable via right-click on the entire statement in the logging console to add a filter for the clicked item. Another useful addition, as stated by the participants, was to apply the filter for the console log, too, so the user could easily review the desired data without being overwhelmed by the rest. A more extensive suggestion was to add "Stories" to the filter feature, meaning filtering for sequences of incoming statements. Many actions in a VR environment are followed by another particular, for example, a button press that is followed by a button release. Adding RegEx to the filter configuration would extend the search capabilities enormously. The last suggestion was to show more details when items like the actor counter or the last statement are clicked.

Reliability and performance

Even though both developers had not yet tested it on their own, both stated their *impression regarding reliability and performance* that LiMoxAPI made a stable impression and especially that ten statements per second could be logged as completely satisfying. More on the technical capabilities in Section 6.2.

Interface

The *usage of the interface* was described as intuitive and self-explaining and the visual appearance as clean and organized, however, with two exceptions. First, the functionality to open a dialog for the complete statement when a statement is clicked in the logging console was stated as not obvious and should be indicated, for example, by a small info text. The other point was the name of the filter component. It was suggested to name the component *counter* instead of *filter* because the main functionality was to count specific statements, and already observed statements were not included in the counter, meaning when the filter is reconfigured during the monitoring process, the counter would continue to increase the current value.

6.1.2 Discussion

The use evaluation confirmed the profit of LiMoxAPI to support developers during the implementation of VR learning environments. The feedback given in the user evaluation was not without criticism, but both developers stated that LiMoxAPI 's provided features were valuable support for the development process. Of course, some of the features still need some improvement, for example, updating the filter with more easily configurable filter options or a more sophisticated filter option in general, as well as some minor changes like more shown details or info for usability. These recommendations are absolutely valid and, more importantly, absolutely feasible, and their possible future implementation will be further described in Future Work (7). But it must be remembered that this was the first confrontation with the user group. For the future, LiMoxAPI needs further evaluations with more participants to cover a broader spectrum of possible issues and improvements.

6.2 Technical Evaluation

The user evaluation showed usability for the target user group, but it does not cover the boundaries of performance and current processing times. This performance test is conducted with the second assessment, the technical evaluation.

6.2.1 The setup

Hardware Spezifications	
OS	Windows 11
CPU	AMD Ryzen 5 2600, 6 cores 3.4 GHz
GPU	Radeon RX 580, 8192 MB VRAM
RAM	16 GB

Table 6.1: The hardware specifications of the hosting test system

The first performance test was to check the average processing time for different data traffic. Therefore, a short rust script was used to upload the different amount of statements per second to the LRS to simulate a VR environment. To check if the uploading frequency was as specified, this VR simulation script was measured as well. The hardware specifications can be seen in Table 6.1. Every phase had configured the same frequency of one second but differed in the amount of statements that were sent with each iteration. The first phase started with one statement per second, the second phase was ten seconds, the third was 100 statements, and the fourth was 1000 statements. The chunk of statements were taken from a data set generated by a VR application developed with the OmiLAXR framework. For each phase, 100 chunks of data were sent to calculate an expressive median after the measurement. For every phase, the average and median were quite similar, but the median had the advantage of disregarding outliers, which were quite common due to computing fluctuations.
The second performance test measured the time it takes for a stored statement to be visualized. For this assessment, the stored timestamp value from ten arrived statements is sent with the remaining payloads to the frontend, where it is then compared with the current timestamp.

6.2.2 Performance results

As stated in the user evaluation, being capable of monitoring ten statements per second without problems is completely fine, and generally, is it unrealistic to send 100 statements per second in a VR learning environment, not even to mention 1000 statements per second, except multiple VR apps that are connected to the same VR are sending statements at the same time. However, the intention for the performance test was not only to check the capacities for a standard throughput but also to test where the boundaries actually are.
The results are plotted in Table 6.1. The horizontal axis shows the number of statements that are sent per second to the LRS, and the vertical axis states the median of the times needed to process the according number of statements in milliseconds (ms). For any realistic number of statements, LiMoxAPI needed only a marginal time. Even 100 statements per second took only about 36 ms, which is about the time one frame lasts in a common cinematic movie [WAH+15]. At 250 statements per second, the pulse monitor started to lag sometimes but was with a median of 114.5 ms, still more than acceptable. However, up to this point, the processing time was more or less insignificantly low, but with over 100 ms, the increase in processing time became noticeable.
The vital problem that arose with it is the longer time period for new statements to arrive. The longer LiMoxAPI needs to process the last new statements, the more statements need to be processed in the next iteration. This vicious circle leads to an exponential growth. The graph shows only a slight exponential growth because the later phases with the bigger chunks were not exactly sending the statements every second but took a little bit longer due to the enormous payload. Even with a multi-threaded rust program, the 1000 statements could only be sent about every 1300 ms, and fluctuations were altering the results as well. At 1000 statements per second, LiMoxAPI had a tough time processing the huge amount of incoming data, and the visualization was constantly lagging. The interface lags are found in the backend. Most of the frontend is only updated with normal-sized data structures like simple integers or arrays. When 1000 new statements are incoming, the pulse monitor array is only extended by one value, the "1000". The latency test resulted in a median of six ms, which is neglectable low. It has to be considered that this is not from the moment when the statement is generated in the VR app but when it arrives in the LRS. This latency is, of course, influenced by the time it takes to process the data, which is the reason the first performance test is the decisive one.

6.2.3 Discussion

The results of the technical evaluation were better than expected. LiMoxAPI proved to handle far more than the usual data traffic. However, it is worth mentioning that the measured times are countable only for the specific hardware setup used for this test 6.1. Nevertheless, it can be expected that the devices used for VR environments generally have high-performance hardware resources.
The reason for the lags is the delay that arises because the backend cannot update the frontend with the configured frequency, and the graph plot frequency is expected to have a much faster rate of new incoming data. The graph's update frequency can be updated, but doing this too often causes lagging as well. So if there is the intention to

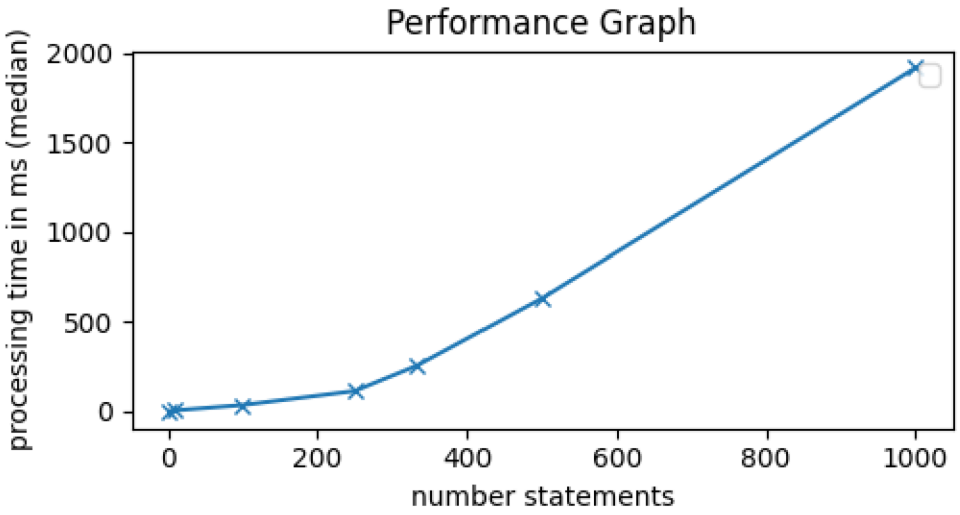


Figure 6.1: The graph shows the different medians of processing time for a specific amount of statements per second.

improve LiMoxAPIs performance, a first solution could be to chunk incoming data and divide the workload into several threads via a threadpool[1] that dynamically creates threads based on the number of incoming statements. Important is that the processed data of all threads is gathered and sent from only one thread so the frontend does not get flooded by multiple threats, each sending data. Another improvement is a sophisticated synchronization between the backend processing frequency and the pulse monitor animation frequency or seeking out a more fitting framework than ApexCharts.js to plot data.
The latency test demonstrated how low the time a statement needs after it arrives in the LRS to be visualized. With six ms, there is no room for apprehension that monitoring an LRS instead of the VR directly causes a disadvantageous delay. Nevertheless, this evaluation showed more than suitable performance qualities for natural usage.

[1] https://docs.rs/threadpool/latest/threadpool/index.html, last accessed: September 2024

Chapter 7 Future Work

During the planning phase, implementation process, and evaluation of LiMoxAPI, many functionalities and features were schemed, discussed, and recommended, but not all of them could be realized yet. Naturally, after the evaluation, many ideas and suggestions came up to improve LiMoxAPI, but only a little time was left, so further development has to be done in the future. In this chapter, future work will be discussed, starting with the section about upgrading the existing features and followed by the discussion about new concepts of LiMoxAPIs architecture.

7.1 Upgrading features

First are, of course, some small but still useful updates that need to be applied. A refresh button should be implemented to remove all captured data without deleting configured filter settings to restart the monitoring process easily. Currently, it is uncomfortable to set up the filter path in the filter feature. One solution would be an autocomplete function that suggests paths based on the user input. For example, when the input field is focused but still empty, it could suggest the highest layer of elements of every statement that is generated with the OmiLAXR framework. When an element is chosen, the next layer of that element is suggested. Another solution, as suggested in the user evaluation, is to create a filter by selecting a specific element or at least copy the path in the full statement view in the logger console of LiMoxAPI. In the first round, the user could capture a couple of statements, display them in full view, and right-click the desired element to filter it for further incoming statements. Enable RegEx for the value field in filter configuration would be a huge profit as well as not only a single value can be filtered but multiple values all of a specific structure for only one configuration.
But the filter generally needs a more extensive upgrade. Currently, the filter configurations are only used to count statements, but it would be useful for the logger console to display only the desired statements. Additionally, the filter does not regard past statements, so when the filter configuration changes, the filter starts from zero and does not display the number of already observed statements fitting the new configuration. This is where LiMoxAPI might need a more profound restructuring. The capable resources of the backend should still do the significant processing, but that would lead to a more complex communication between both layers to provide the correct data for the backend. Contributing to this significant change is the suggested feature to filter for sequences of statements. These improvements require storing the statements that demand a new concept for the data flow. It must be evaluated if past monitored

statements should also be considered or if LiMoxAPI should stay a pure live monitoring tool.

7.2 Improving LiMoxAPIs architecture

In the last section were already mentioned some changes to the technical composition of LiMoxAPI regarding the filter feature. In this section, deeper technical improvements in the future are presented.

7.2.1 Integrating the LRS

LiMoxAPI connects to an external LRS that receive the generated statements of a VR learning environment. Using the preconfigured SQL LRS from Yet Analytics is recommended as it can be used locally easily without any configuration effort. However, it still needs to be downloaded unzipped and started every time LiMoxAPI gets started. Therefore, a huge improvement would be an integration of an LRS into LiMoxAPI. That could be by just adding the configured SQL LRS to the downloadable LiMoxAPI package and adding the corresponding launch functionality to the rust backend. Another more complex approach is implementing a custom LRS in the backend[1]. The rust library Actix Web[2] could provide the Rest API and Diesel.rs[3] SQLite queries. It has to be evaluated if it is worth the effort because setting up a custom LRS is quite complex [XAP24]. The first approach will be more suitable as it is far faster to realize.

7.2.2 Modularization of features via plug-ins

The current features of LiMoxAPI like the pulse monitor are functions that expect an array of statements to process it before the resulting information is sent to the according frontend component. These functions are called in the main loop. This gives the developer of LiMoxAPI the possibility to add a new function, but it has to be implemented without any existing structure, and how to update the frontend is coded individually. A standard here would help specify how developers can easily provide their own features for LiMoxAPI. One approach is to implement a feature rust trait[4] with the necessary functions to process the passed data and send it to a specific frontend component using the tauri IPC. The same goes for the frontend with a JavaScript class that provides a communication function to receive data from a particular backend component. With these two structures, developers could implement their own features and provide them packaged together for LiMoxAPI. To go a step even further, it would also enable the user of LiMoxAPI to select which feature to include during runtime. Providing commands for the command-line interface would facilitate the integration of these plug-ins. This could even lead to a sophisticated package manager system that enables developers to build open-source LiMoxAPI features that are downloadable via the package manager.

[1] https://xapi.com/building-a-learning-record-store/, last accessed: September 2024
[2] https://actix.rs/, last accessed: September 2024
[3] https://diesel.rs/, last accessed: September 2024
[4] https://doc.rust-lang.org/book/ch10-02-traits.html, last accessed: September 2024

Chapter 8 Conclusion

This project aimed to provide a monitoring tool to developers who realize VR learning environments with the xAPI specification. One major problem was the challenge to check if xAPI statements were generated correctly in the VR application and successfully sent to the connected Learning Locker. This LRS provided by the research group Learning Technologies is not always available and does not provide the sophisticated functionalities as needed. LiMoxAPI was developed in this bachelor thesis scope to address this problem by live monitoring an LRS that could be any local variant.
LiMoxAPI is a desktop application developed with the framework Tauri that uses Rust in the backend and JavaScript or plain HTML & CSS in the frontend for the visualization. The main views are the connection panel to connect to an LRS and the dashboard where the features are visualized. The dashboard consists of several components, including a pulse monitor, counters, and a logging console. Amongst others, these features resulted from the requirement analysis and were evaluated in the end with a user and technical evaluation. As stated in the user evaluation, these assessments demonstrated the profit of LiMoxAPI to the development process. The technical evaluation also showed that the monitoring tool performed better than necessary.
However, reasonable improvements for LiMoxAPI still exist. The assessment resulted in further functionalities and updates of existing features expedient for LiMoxAPI that could not be implemented due to time constraints. This includes an improved filter feature to filter statements in the loggger console and to configure more sophisticated filter options. Another vision for the future is an integration of the LRS into LiMoxAPI and a plug-in system that allows developers of LiMoxAPI to integrate new features more easily. Improving the performance is a less urgent matter but could be realized by separating the data processing into threads. However, receiving more than 100 statements per second is not a natural scenario. Overall, LiMoxAPI is an already practical live monitoring tool but has room for further improvement.

Appendix A Bibliography

[BL21] J. Z. Blanco and D. Lucrédio. A holistic approach for cross-platform software development. *Journal of Systems and Software*, 179:110985, September 2021. doi:10.1016/j.jss.2021.110985.

[BV17] Leif P. Berg and Judy M. Vance. Industry use of virtual reality in product design and manufacturing: A survey. *Virtual Reality*, 21(1):1–17, 2017. doi:10.1007/s10055-016-0293-9.

[CDST12] Mohamed Amine Chatti, Anna Lea Dyckhoff, Ulrik Schroeder, and Hendrik Thüs. A reference model for learning analytics. *International journal of Technology Enhanced learning*, 4(5-6):318–331, 2012.

[con24] Tauri contributors. Tauri 2.0. https://v2.tauri.app/, 2024, last accessed: September 2024.

[CRHURCC22] Leonor Adriana Cárdenas-Robledo, Óscar Hernández-Uribe, Carolina Reta, and Jose Antonio Cantoral-Ceballos. Extended reality applications in industry 4.0.–a systematic literature review. *Telematics and Informatics*, 73:101863, 2022.

[FCV+21] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *USENIX Symposium on Usable Privacy and Security (SOUPS)*, 2021.

[Fou24] OpenJS Foundation. Electron. https://github.com/electron/electron, August 2024, last accessed: September 2024.

[GHS24] Sergej Görzen, Birte Heinemann, and Ulrik Schroeder. Towards using the xAPI specification for Learning Analytics in Virtual Reality. Kyoto, Japan, 2024.

[HEGS22] Birte Heinemann, Matthias Ehlenz, Sergej Görzen, and Ulrik Schroeder. xapi made easy: A learning analytics infrastructure for interdisciplinary projects. *International Journal of Online & Biomedical Engineering*, 18(14), 2022.

[HGS+22] Birte Heinemann, Sergej Görzen, Ulrik Schroeder, J Bourdin, and E Paquette. Repix vr-learning environment for the rendering pipeline in virtual reality. *Bourdin, J.-J.; Paquette, E., Hrsg.: Eurographics*, 2022.

[HGS23] Birte Heinemann, Sergej Görzen, and Ulrik Schroeder. Teaching the basics of computer graphics in virtual reality. *Computers & Graphics*, 112:1–12, 2023.

[HPH+17] Tobias Huber, Markus Paschold, Christian Hansen, Tom Wunderling, Hauke Lang, and Werner Kneist. New dimensions in surgical training: Immersive virtual reality laparoscopic simulation exhilarates surgical staff. *Surgical Endoscopy*, 31(11):4472–4477, November 2017. doi:10.1007/s00464-017-5500-6.

[Hus18] Anealka Aziz Hussin. Education 4.0 made simple: Ideas for teaching. *International Journal of Education and Literacy Studies*, 6(3):92–98, 2018.

[Ini24] Advanced Distributed Learning Initiative. xAPI-Specification. https://github.com/adlnet/xAPI-Spec, September 2023, last accessed: September 2024.

[ISO11] ISO-Norm. ISO/IEC DIS 25010(en), Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:dis:ed-2:v1:en, 2011.

[Jav24] JavaScript framework benchmark. https://krausest.github.io/js-framework-benchmark/2024/table_chrome_128.0.6613.86.html, 2024, last accessed: September 2024.

[Lam21] Nebrass Lamouchi. *Introduction to the Monolithic Architecture*, pages 35–38. Apress, Berkeley, CA, 2021. doi:10.1007/978-1-4842-7170-4_2.

[Lea24a] LearningLocker/deploy. https://github.com/LearningLocker/deploy, May 2024, last accessed: September 2024.

[Lea24b] Learn Svelte. https://v2.svelte.dev/guide, last accessed: September 2024.

[MGMADGM17] Jorge Martín-Gutiérrez, Carlos Efrén Mora, Beatriz Añorbe-Díaz, and Antonio González-Marrero. Virtual technologies trends in education. *Eurasia journal of mathematics, science and technology education*, 13(2):469–486, 2017.

[MM07] Roxana Moreno and Richard Mayer. Interactive multimodal learning environments: Special issue on interactive learning environments: Contemporary issues and trends. *Educational psychology review*, 19:309–326, 2007.

[MMR14] Jose P. Miguel, David Mauricio, and Glen Rodriguez. A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications*, 5(6):31–53, November 2014. arXiv:1412.2977, doi:10.5121/ijsea.2014.5603.

[Nah22] Dotan Nahum. Tauri vs. Electron for Tray Apps. https://betterprogramming.pub/tauri-vs-electron-for-tray-apps-ed15974f35ce, November 2022.

[Ove24] Stack Overflow. Technology | 2024 Stack Overflow Developer Survey. https://survey.stackoverflow.co/2024/technology#most-popular-technologies, 2024, last: accessed: September 2024.

[Pre05] Roger S Pressman. Software engineering: a practitioner's approach. *Pressman and Associates*, 2005.

[RMFW20] Jaziar Radianti, Tim A Majchrzak, Jennifer Fromm, and Isabell Wohlgenannt. A systematic review of immersive virtual reality applications for higher education: Design elements, lessons learned, and research agenda. *Computers & education*, 147:103778, 2020.

[Sak19] E. Saks. JavaScript Frameworks: Angular vs React vs Vue. 2019.

[Sof24] Rustici Software. Learning Record Store. https://xapi.com/learning-record-store, March 2024, last accessed: August 2024.

[Sou12] Tiago Boldt Sousa. Dataflow Programming Concept, Languages and Applications. 2012.

[Ude21] Udesh S Senarath. Waterfall Methodology, Prototyping and Agile Development. 2021. doi:10.13140/RG.2.2.17918.72001.

[Vue24] Vue.js - Reactivity in Depth. https://vuejs.org/guide/extras/reactivity-in-depth.html, last accessed: September 2024.

[Vya22] Rishi Vyas. Comparative Analysis on Front-End Frameworks for Web Applications. *International Journal for Research in Applied Science and Engineering Technology*, 10(7):298–307, July 2022. doi:10.22214/ijraset.2022.45260.

[WAH+15] Laurie M. Wilcox, Robert S. Allison, John Helliker, Bert Dunk, and Roy C. Anthony. Evidence that Viewers Prefer Higher Frame-Rate Film. *ACM Transactions on Applied Perception*, 12(4):1–12, September 2015. doi:10.1145/2810039.

[XAP24] xAPI - Technical. https://xapi.com/developer-overview/, 2024, last accessed: September 2024.

Appendix B Assessment Questionary

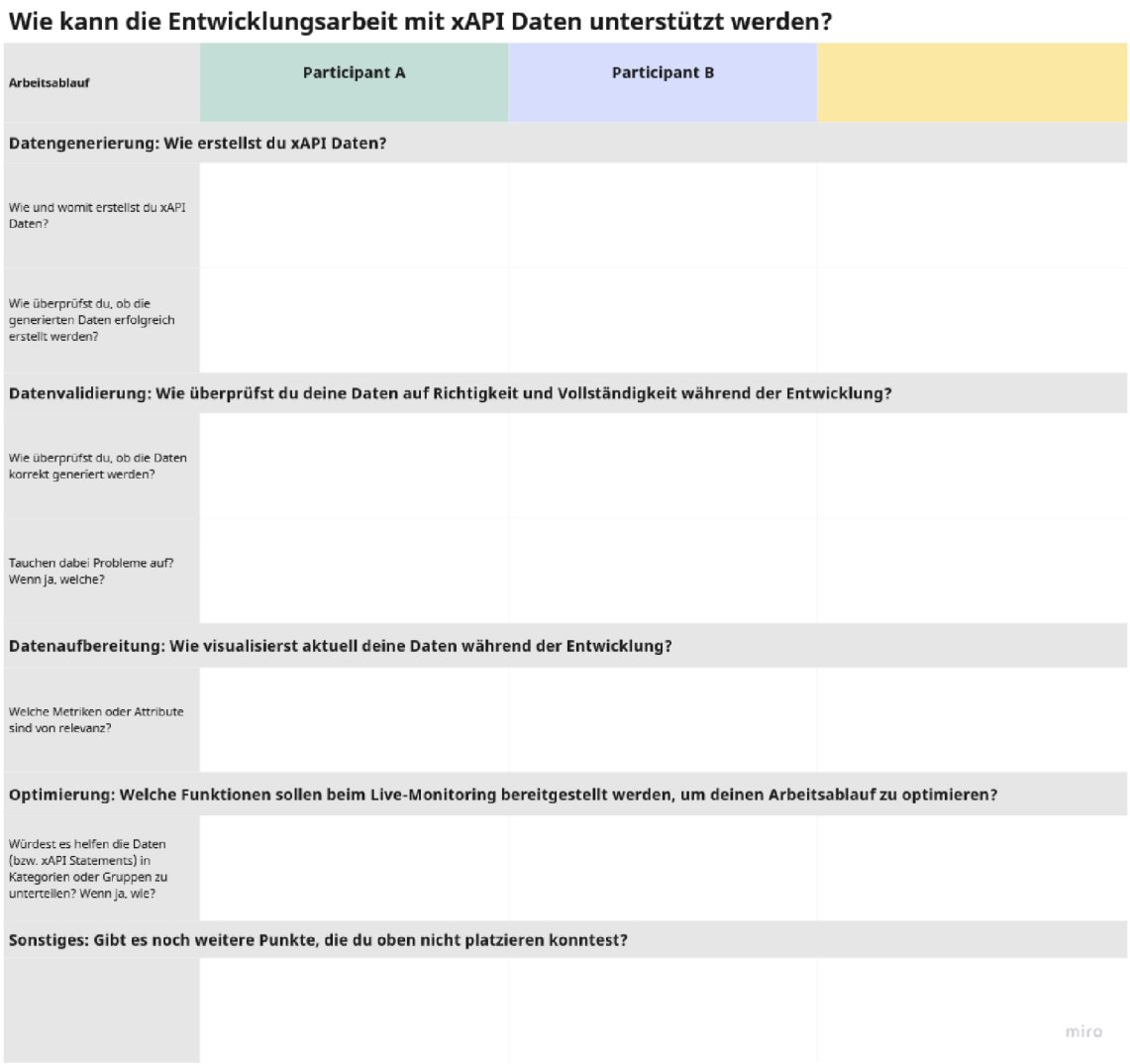


Figure B.1: The questionary for the requirement analysis

LiMoxAPI - Evaluation

Arbeitsablauf	Participant A	Participant B	
Funktionalität			
Sind die bereitgestellten Funktionen hilfreich/sinnvoll			
Gibt es Funktionalitäten, die du vermisst?			
Zuverlässigkeit und Performance			
Konntest du LiMoxAPI zuverlässig nutzen (z.B. keine unerwarteten Verbindungsabbrüche)?			
Wie hast du die Performance empfunden? War die Laten, bis ein Statement in LiMoxAPI eingetroffen ist, annehmbar.			
Usability			
Wie intuitiv war die Nutzung von LiMoxAPI?			
Wie hast du die visuelle Aufmachung empfunden?			
Gab es Dinge, die du irreführend fandest?			
Sonstiges: Gibt es noch weitere Punkte, die du oben nicht platzieren konntest?			

miro

Figure B.2: The questionary for the user evaluation

Glossary

JSON	The JavaScript Object Notation is data format to store and transmit data.
RegEx	A regular expression is a sequence of characters that defines a match pattern for a text.
SQL LRS	SQL LRS is the Learning Record Store of YetAnalytics.

Acronyms

IPC	Inter-Process Communication.
LA	Learning Analytics.
LiMoxAPI	Live Monitoring of xAPI statements.
LRS	Learning Record Store.
ms	milliseconds.
OmiLAXR	Open and modular integration of Learning Analytics in XR.
ReCoPA	Researcher Companion Panel.
VR	Virtual Reality.

Acknowledgement

First of all, I want to thank my supervisor, Sergej Görzen, for his support and feedback during this time. I also want to thank Ana and David, who took their time to discuss the project and evaluate the result. Last but not least, I want to thank the research group Learning Technologies for giving me this opportunity to write my bachelor's thesis.