A PROJECT REPORT

ON

THE DESIGN AND IMPLEMENTATION OF AN AUTOMATED GAIN STAGING AND VOLUME MATCHING SYSTEM FOR REAPER

BY

JEHOVY LEAD OBARO

PSC2105350

PRESENTED TO

THE DEPARTMENT OF COMPUTER SCIENCE,

FACULTY OF PHYSICAL SCIENCE, UNIVERSITY OF BENIN,

BENIN CITY, EDO STATE,

NOVEMBER 2025

THE DESIGN AND IMPLEMENTATION OF AN AUTOMATED GAIN STAGING AND VOLUME MATCHING SYSTEM FOR REAPER

BY

JEHOVY LEAD OBARO

PSC2105350

SUBMITTED TO

THE DEPARTMENT OF COMPUTER SCIENCE, FACULTY OF PHYSICAL SCIENCES, UNIVERSITY OF BENIN, IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF BACHELOR OF SCIENCE (B.Sc.) DEGREE COMPUTER SCIENCE

NOVEMBER 2025

CERTIFICATION

This is to certify that this project work was carried out by JEHOVY LEAD OBARO with Matriculation Number PSC2105350 under my supervision. It is adequate and satisfactory, both in scope and content, for the award for the Bachelor of Science (B.sc) Degree in Computer Science of the University of Benin.

_____ _____

MR. B. J. ODETAYO **DATE**

Project Supervisor

APPROVAL

This project work is hereby approved in partial fulfilment of the requirements for the award of Bachelor of Science (B.Sc) Degree in Computer Science from the University of Benin

PROF. (MRS.) R. O. USIOBAIFO	**DATE**
Head of Department	

DEDICATION

This project work is dedicated to God, foremost, and to my parents, who sacrifice daily to ensure I achieve academic success; along with every mind in the open-source communities and developers who guided my path towards this incredibly niche path in audio computing.

ACKNOWLEDGEMENT

I express my sincere gratitude to God for the support and guidance I received throughout my academic journey. My deepest appreciation goes to my project supervisor, Mr B. J. Odetayo, for his methodical approach towards ensuring every of his assigned project students get carried along till the end.

I would also like to specially thank the Head of the Department of Computer Science, Prof. (Mrs.) R. O. Usiobaifo, for leading every staff and student towards academic success.

I am also thankful to my lecturers for providing the strong academic foundation that prepared me for this project:: Prof. (Mrs.) V.I. Osubor, Dr. E.C. Igodan, Dr. (Mrs.) L.O. Usiosefe, Mr. J. Okhuoya, Mr. I.E. Obayagbonna, Dr. (Mrs.) Aziken, Prof. G.O. Ekuobase, Dr. F.O. Oliha, Prof. (Mrs.) V.V.N. Akwukwuma, Prof. F.I. Amadin, Dr. F.O. Chete, Dr. (Mrs.) R.O. Osaseri, Mr. K.O. Otokiti, Dr. E. Nweli, and Mr. D.N. Idehen.

To my friends and fellow project mates, thank you for your persistence and collaboration. I was impacted immensely.

A special and heartfelt thank you to my parents, Mr. and Mrs. Jehovy. Your unwavering care, encouragement and generous financial support were essential to my success.

I am truly grateful for you all.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

During audio mixing and mastering, particularly when using equalization and compression, manually matching the perceived volume levels before and after plugin processing is a critical yet time-consuming task. This manual process involves repeatedly analyzing audio segments to ensure volume stability, which is crucial for a balanced mix. This project aims to automate this volume matching process. The proposed solution is **a Reaper Lua script (potentially incorporating JSFX for analysis) that reads both the input and output audio levels of a track or effect chain. It will then automatically balance the output audio's loudness to match the input**, offering the user choices for the measurement standard, such as LUFS, RMS, or peak dB, thus ensuring efficient and accurate gain staging.

CHAPTER ONE

INTRODUCTION

1.1 Background of the Study

In digital audio production, achieving a consistent and controlled loudness level is one of the most important aspects of mixing and mastering. Loudness consistency affects how individual tracks blend with each other, how the mix translates across playback systems, and how the final song is perceived by the listener (Katz, 2015; Izhaki, 2017). Even when technical levels are correct, perceived loudness i.e. what the human ear interprets as "volume"; can still vary due to dynamics, frequency content, and processing (Vickers, 2010).

One common workflow in mixing is applying various audio effects such as equalizers (EQs), compressors, saturators, and other dynamic processors to enhance or correct a sound. These plugins, however, often affect the loudness of the signal in unintended ways. For instance, a compressor can reduce the overall level of a track by controlling its dynamics, while an EQ can increase volume if it boosts key frequency ranges (Senior, 2011). These changes can make it harder for the mixing engineer to judge whether the effect itself is improving the sound, or if the perceived improvement is simply due to a change in loudness (Huber and Runstein, 2017).

This challenge becomes even more apparent when bypassing a plugin to compare the processed signal with the original. A plugin might make a track louder or quieter, which skews the listener's perception. The louder version often sounds better, even if it's not technically superior (Katz, 2015). This false impression can mislead decision-making and ruin carefully constructed gain staging and balance. Many audio engineers work around this by manually adjusting output gain or using gain compensation plugins, but this process is tedious, time-consuming, and error-prone (Izhaki, 2017).

To solve this problem, this project introduces an automated solution designed within the REAPER digital audio workstation (DAW). REAPER is uniquely positioned for this kind of innovation because it allows users to write custom scripts in Lua (via ReaScript) and create custom audio processing plugins using JSFX (Jesusonic Effects). These tools provide deep control over the DAW environment, allowing tailored solutions to real-world audio problems.

The system being developed in this project listens to the loudness of a track measured in LUFS-I (Integrated Loudness Units Full Scale) before and after a plugin is bypassed. It then calculates the difference and automatically adjusts the volume using a gain plugin to restore loudness consistency. This ensures that bypassing any plugin, whether for A/B testing or workflow purposes, does not disrupt the overall balance of the mix (EBU, 2014).

The inspiration for this solution came from personal mixing sessions, where plugins like EQs or compressors inserted after careful volume balancing would introduce unwanted volume shifts when toggled on or off. These shifts often destabilized the relative loudness across the mix, requiring tedious manual gain adjustments. Building a tool to automate this process not only improves personal workflow but also has the potential to help other REAPER users facing the same issue.

1.2 Statement of the Problem

In audio production, plugin bypassing is commonly used to compare the original and processed versions of a sound. However, many audio effects such as compressors, EQs, and saturators alter the loudness of a signal. When these plugins are bypassed, the resulting volume difference can mislead the listener into thinking the change in sound quality is due to the processing, when in fact it’s just a difference in loudness (Katz, 2015; Giannoulis et al., 2012).

This disrupts accurate A/B comparisons, slows down workflow, and undermines careful gain staging. Currently, users must manually adjust output gain to match perceived loudness before and after bypass, which is both time-consuming and prone to error.

To the best of my knowledge, there is no built-in tool in REAPER that automates this task. This project addresses that gap by developing a system that detects loudness differences when a plugin is bypassed and automatically compensates for them. This maintains a consistent volume for more accurate mixing decisions.

1.3 Aims and Objectives of the Study

The aim of this project is to develop an automated system within REAPER that maintains consistent perceived loudness when bypassing audio plugins. The system aims to achieve the following objectives:

1. Investigate and analyse the challenges of the existing method of gain matching.
2. Design and develop a time-based gain matching software that measures loudness, calculates the difference and adds the volume to the bypassed plugin.
3. Evaluate and test the system.
4. Deploy the Auto Gain Staging and Volume matching system.

1.4 Significance of the Study

With the growing importance of consistent perceived loudness in professional audio mixing and mastering, especially when using plugin chains, this REAPER-based gain staging solution will help achieve the following:

1. It will reduce the problem of inaccurate A/B comparisons caused by loudness jumps when bypassing plugins, thereby allowing for more objective mix decisions.
2. It will improve workflow efficiency by reducing the need for manual gain matching, saving time for engineers and reducing ear fatigue.
3. It will serve as a template for integrating automation and scripting within REAPER, encouraging more engineers to explore its customizable environment.

1.5 Scope and Limitations

This project focuses on developing a plugin-based solution within REAPER using Lua scripting and JSFX to automate gain compensation during plugin bypass. It targets audio effects that alter perceived loudness, using LUFS-I as the measurement standard (ITU-R BS.1770-4, 2015). The system is intended for standard track setups in mixing and mastering, excluding complex routings and parallel chains.

This study is limited to standard track setups in REAPER, excluding complex routings and parallel processing chains, which may require different approaches to gain compensation. The system's performance may also depend on the processing strength of the user's hardware and the health of the specific plugins used, which could introduce variability in real-time processing.

CHAPTER TWO

LITERATURE REVIEW

2.1 Review of Related Work, Theories, and methods

2.1.1 Overview of Audio Gain Staging

Gain staging is the process of managing signal levels at different points in an audio chain to ensure clarity, avoid distortion, and maintain control over dynamics (Senior, 2011; Izhaki, 2017). In simple terms, it's about making sure your audio isn't too loud or too soft as it passes through different devices or plugins, whether that's an EQ, compressor, reverb, or even meters. This technique is especially important when dealing with digital audio workstations (DAWs), where pushing levels too high can cause clipping, while too-low levels can introduce unnecessary noise or make it harder to process tracks properly (Huber & Runstein, 2017).

In Reaper, and other DAWs, gain staging happens both visually and sonically; you listen, and you look at meters. When you're working with multiple plugins in a chain, it becomes crucial to make sure the signal entering and leaving each one stays within an optimal range. That's what gain staging is all about: control, consistency, and clean sound.

Importance of Consistent Loudness in Mixing

A well-balanced mix is built on consistent loudness. When you gain stage properly, it becomes easier to judge EQ changes, compression, or saturation decisions without being distracted by volume shifts. This matters because human ears often perceive louder as better even if the actual quality of the audio hasn't improved (Moylan, 2014). So if, for example, a plugin adds gain after processing, it might trick you into thinking your mix sounds better, when in reality, it's just louder.

This becomes a real problem when you've already spent time volume balancing your tracks; setting each instrument or vocal in its correct relative place. A plugin that changes volume after processing can undo that balance. You lose the mix's integrity. That's where LUFS (Loudness Units relative to Full Scale), specifically **LUFS-I** (Integrated LUFS), comes in. LUFS-I helps measure loudness over time, giving you a stable, average reading (ITU, 2015). That's what I used in my JSFX plugin so that even when changes happen, I can monitor and adjust the volume to keep everything balanced.

Role of Gain Staging in Plugin Chain

Plugins are where gain staging either thrives or falls apart. EQ boosts can subtly increase loudness. Compression often reduces it. Some saturators or limiters can inflate perceived loudness even more. When multiple plugins are stacked in a chain, the combined level changes can quickly get out of hand (Izhaki, 2017).

This is where auto gain staging becomes helpful. It brings the processed signal back to match the original loudness before the plugin was applied. For example, in mixing, if you apply compression to a snare track that was already perfectly volume-balanced with the kick, you don't want that compression to make the snare disappear in the mix;or worse, jump out too much. Similarly, during mastering, every little gain shift affects the overall perceived loudness and tone (Katz, 2015) .

By using LUFS-I in my plugin, I can keep the perceived loudness consistent whether I'm EQing, compressing, or applying other effects. The aim is simple: let the engineer hear the true effect of a plugin, without the distraction of unexpected volume jumps (Senior, 2011) . Whether it's on individual tracks, buses, or the master channel, gain staging is the glue that keeps the entire signal flow honest.

2.1.2 Manual Gain Matching Techniques

Manual gain matching is one of the most common ways audio engineers try to maintain consistent loudness when working with plugins. It refers to the process of adjusting the

output gain of a plugin to match the perceived loudness of the input signal either by ear or by watching meters (Senior, 2011).

Bypass and Level Matching by Ear

One of the simplest methods involves toggling a plugin on and off while adjusting its output gain until the processed signal sounds about as loud as the unprocessed one. This is known as "bypass level matching." It's widely used in mixing because our ears tend to interpret louder sounds as better (Moylan, 2014). Without proper matching, an engineer might think a plugin is improving the sound, when it's just making it louder. However, this method is subjective and can be affected by fatigue, mood, or even room acoustics.

Using Meters to Balance Input/Output

To reduce subjectivity, many engineers use meters (VU meters, peak meters, and more recently LUFS meters) to measure and compare the loudness levels before and after a plugin. LUFS, which stands for Loudness Units relative to Full Scale, is especially useful because it considers how humans perceive loudness over time (EBU, 2011). Matching LUFS readings at the input and output of a plugin provides a more objective approach than relying on ears alone (Katz, 2015).

Limitations of Manual Process

While manual methods offer flexibility, they're often time-consuming and can interrupt workflow especially when working with many plugins across multiple tracks. Constantly toggling bypass, reading meters, and adjusting output gains can add up. There's also the risk of human error or misjudgment, particularly when working under pressure or in less-than-ideal listening conditions. These limitations are what inspired the development of my JSFX plugin, which aims to automate gain matching using LUFS-I as a reference, allowing for more stable and efficient mixing sessions.

2.1.3 Gain Compensation in Audio Plugins

Many commercial audio plugins now include built-in auto gain or gain compensation features to help users maintain consistent loudness throughout their signal chain. This feature detects changes in perceived loudness caused by a plugin and attempts to adjust the output gain to match the input level automatically. While the idea is to save time and avoid bias from louder sounds, implementation varies widely between plugin manufacturers (Katz, 2015; Giannoulis et al., 2012).

How Commercial Plugins Implement Auto Gain

Auto gain is usually based on internal loudness measurements. Some plugins track RMS (Root Mean Square) levels, while more advanced ones use LUFS or perceptual algorithms (ITU-R BS.1770-4, 2015). The goal is to make the processed signal sound as loud as the unprocessed one, making it easier to judge the actual tonal or dynamic effect of the plugin (Moylan, 2014).

Some modern plugins incorporate transparent and user-centered approaches to gain compensation, automatically adjusting output levels to match the input signal's loudness. For instance, certain compressors adjust the output as compression is applied, helping the user focus on tonal shaping rather than being distracted by volume jumps (Giannoulis et al., 2012). This implementation is subtle and designed not to skew A/B judgments, which is critical for accurate mixing decisions.

Some plugins employ simpler forms of gain compensation, often relying on RMS detection or fixed-level makeup gain. For example, certain bus compressors apply automatic makeup gain based on threshold and ratio settings, prioritizing analog-style behavior and simplicity over deep loudness analytics (Senior, 2011).

Some advanced plugins incorporate intelligent tools that analyze the incoming signal's loudness and suggest gain compensation based on spectral balance and target loudness. These systems often leverage digital signal processing techniques to assist with

context-aware decisions, making them particularly useful for mastering and complex multi-track mixes (Zölzer, 2011).

2.1.4 Existing REAPER Scripts Related to Gain Matching

REAPER's user community has contributed various scripts and JSFX tools to enhance the digital audio workstation's functionality, some of which address gain staging or volume management. For example, external VST plugins exist that automatically adjust gain to maintain consistent levels, though they are often focused on specific tasks like vocal riding and are not open-source. Additionally, several JSFX utilities and Lua scripts allow for LUFS-based normalization or gain adjustments, but these typically operate as batch or item-based processes rather than in real-time. Other tools provide loudness analysis or envelope-based adjustments, yet they lack dynamic, plugin-specific compensation within live FX chains.

2.1.5 Psychological Impact of Loudness on Perceived Quality

In the world of audio production, perception is everything and loudness is one of its most deceptive factors. When presented with two identical pieces of music, listeners will consistently prefer the louder version. This phenomenon, widely documented in psychoacoustics, is referred to as **loudness bias** (Moylan, 2014). It plays a powerful role in how both casual listeners and professional engineers assess sound quality.

The Loudness Bias in Everyday Mixing

You've likely experienced it firsthand: you're testing a new plugin or a different EQ curve, and you think the track suddenly sounds "better", more exciting, clearer, more powerful. But what really happened? Most likely, the gain increased by a few decibels, tricking your brain into hearing improvement where there was none.

The psychology is simple but impactful: **the human ear perceives louder as better**. This is due to how our auditory system responds to energy. At higher amplitudes, frequencies,

especially in the midrange, tend to appear more detailed. Low and high frequencies can feel more extended, and transients seem to hit harder (Vickers, 2010). It's a cognitive bias baked deep into how we perceive audio stimuli.

According to studies in psychoacoustics, even a 0.5 dB increase in loudness can influence perceived quality. Experiments have shown that listeners often choose the louder version of a track as sounding "better," even when it's the same mix with only a subtle gain boost (Moylan, 2014). This aligns with the Equal Loudness Contour, which shows that the ear responds differently to different frequencies at various loudness levels. At lower volumes, our ears become less sensitive to bass and treble, which can make mixes feel flat or dull unless compensated.

The Loudness Wars

One of the most infamous demonstrations of loudness bias is the **Loudness Wars**, a decades-long period in which record labels, artists, and mastering engineers competed to make their music louder than anyone else's. It began in earnest in the late 1990s and peaked around the 2000s, with albums being pushed to the brink of digital clipping and dynamic range collapse (Vickers, 2010).

Albums like Metallica's *Death Magnetic* (2008) became infamous not for their musical content, but for their crushed dynamics and distorted audio, a direct result of aggressive loudness normalization. The irony? Louder tracks sounded worse over time, especially on consumer playback systems. The push for loudness sacrificed depth, emotion, and dynamics, all because louder initially "felt better."

Eventually, the industry began to push back. The rise of **streaming platforms** like Spotify, Apple Music, and YouTube which use **loudness normalization algorithms** helped neutralize the advantage of louder masters. These platforms now reduce the volume of overly loud tracks to meet consistent playback targets, such as those recommended for streaming (EBU, 2014). This marked a cultural shift: instead of rewarding volume, we began to reward quality again.

2.1.6 Related Works

Automated & Academic Systems for Gain Staging

1. Recent academic work has explored automated gain staging solutions. For example, Ward et al. (2023) developed techniques for modeling partial loudness and masking in multi-track mixes, enabling clearer automation-based balancing that optimizes loudness across tracks (Ward et al., 2023).
2. That study inspired later advances; **Ward et al.** built on that work, focusing on modeling partial loudness and masking in multi-track mixes for clearer automation-based balancing (Ward et al., 2023).

These academic systems reliably produce consistent loudness across tracks based on predefined targets. However, they operate offline or batch-wise, not in real-time within a DAW session.

Open-Source & Community Tools in REAPER

Within REAPER's ecosystem, some community-built tools partially address loudness workflows. Scripts and utilities exist for LUFS-based normalization across items, loudness metering, and batch processing actions. However, these tools typically operate as offline processes, lacking real-time response or dynamic context awareness for plugin-specific compensation.

2.2 Critical Analysis of Previous Studies

2.2.1 Limitations of DAWs Without Built-in Gain Compensation

While modern Digital Audio Workstations (DAWs) come with a wide array of mixing tools, many still lack native support for automatic gain compensation, a feature that can significantly improve workflow and mixing accuracy (Izhaki, 2017). REAPER, for instance, is a highly customizable and powerful DAW, but it doesn't offer built-in auto gain matching on a plugin or track level. This means that every gain-related change in an effect

like compression or EQ must be manually matched by the user to avoid loudness bias (Senior, 2011).

The absence of automatic gain matching makes critical listening more difficult, especially during fast-paced mixing sessions. Users often have to toggle bypass repeatedly or rely on loudness meters to try and balance input/output levels. This interrupts creative flow and can result in misjudging the effect of a plugin, simply because the louder version feels better to the ear (Katz, 2015). Over time, this adds up to less consistent mixes, more ear fatigue, and a heavier cognitive load on the engineer.

Without built-in compensation, users must either insert dedicated metering tools before and after each plugin or trust their ears in potentially unreliable environments. For beginner and even intermediate mixers, this limitation can become a hidden bottleneck, preventing them from achieving more transparent and intentional sound design (Izhaki, 2017). It's precisely this gap that a JSFX-based gain compensation tool seeks to fill, which is bringing accuracy, efficiency, and confidence back into the mixing process.

2.2.2 Limitations of Existing REAPER Scripts

Despite these offerings, none of the existing solutions provide a real-time, integrated gain compensation system that:

1. Measures LUFS-I (integrated loudness) before and after an FX plugin.
2. Automatically adjusts output gain to match the original loudness.
3. Can be applied dynamically across tracks, buses, or the master channel.
4. Is designed to remain transparent and preserve the relative balance in complex mixes.

This gap highlights the necessity for a tool like mine,designed from the ground up for real-time LUFS-based gain compensation within REAPER. By embedding this into a JSFX plugin, users can maintain loudness consistency through various mixing and mastering

processes without disrupting prior gain staging or introducing bias during plugin evaluations.

2.2.3 Limitations of Plugin-Based Systems

Despite their usefulness, built-in auto gain features have limitations. Some only function effectively in specific modes or require manual adjustments to achieve accurate results (Giannoulis et al., 2012). Others may not account for rapid transients or dynamic shifts, leading to inconsistent results. Additionally, since each plugin handles gain differently, using multiple plugins in a chain can still result in unpredictable loudness levels. This inconsistency highlights the need for a more centralized and customizable solution such as a dedicated JSFX gain matching tool based on LUFS-I that can sit anywhere in the signal chain (ITU-R BS.1770-4, 2015).

2.2.4 Existing Plugins with Static or Manual Gain Matching

Other tools exist across platforms, such as LUFS meters that measure loudness and suggest gain offsets. However, their integration into DAW automation is often manual rather than reactive, limiting their ability to dynamically compensate for plugin-induced loudness changes (Katz, 2015).

2.3 Justification for the Approach

2.3.1 The Need for Objective Loudness Comparison

In modern audio production, **gain matching tools** and **auto loudness compensation plugins** have become vital. They ensure that any adjustments (compression, saturation, EQ) are evaluated fairly. Without them, it’s nearly impossible to make accurate A/B comparisons. You might prefer the processed version not because it’s more musical, but because it's louder (Senior, 2011).

Many modern plugins now offer integrated loudness matching, making it easier to focus on actual sonic differences rather than perceptual tricks (Katz, 2015).

2.3.2 Human Bias vs. Technical Accuracy

Still, loudness bias is so ingrained in human perception that even trained engineers can fall victim to it. This underlines the importance of creating tools that **remove the human bias from the decision-making chain**. If we can ensure volume parity between plugin states or processing chains, then decisions become about tone, texture, and timing; not about which is louder.

Gain matching doesn't just improve technical accuracy; it protects creative integrity. It allows you to trust what you're hearing (Moylan, 2014).

2.3.3 The Need for Workflow Automation in Mixing

Modern music production is no longer confined to the walls of million-dollar studios or large teams of engineers. With the democratization of audio tools and the rise of bedroom producers, the demand for speed, efficiency, and consistent quality has grown exponentially (Izhaki, 2017). In this context, workflow automation has emerged as one of the most transformative forces in the mixing process, changing not just *how* music is mixed, but *who* can mix, *when*, and *how quickly* they can achieve professional results.

Time-Saving Benefits

Mixing is often romanticized as a purely creative endeavor, but any experienced engineer knows that it's filled with time-consuming technical steps: gain staging, plugin chain balancing, A/B testing, level matching, session routing, repetitive volume automation, and so on (Izhaki, 2017). These repetitive tasks can eat up hours in a session, especially when managing large, multi-track projects.

Automation tools, whether built into DAWs or developed via custom scripts and macros, can perform these tasks in seconds. For example, a script that automatically sets consistent

levels across all vocal takes, or a plugin that matches pre/post gain levels when applying compression, can save significant time per track. Over the course of a full mix session, that time savings compounds significantly.

Cognitive Load and Fatigue

Mixing is an intensely detail-oriented process. It demands hundreds of micro-decisions: EQ moves, compressor settings, stereo placement, automation curves. When an engineer also has to constantly monitor gain changes, match levels, or reset plugin outputs manually, the mental fatigue can pile up fast. By offloading repetitive or low-level tasks to automated systems, engineers preserve mental bandwidth for more critical thinking like judging tonal character, balancing emotional impact, or sculpting dynamics with nuance. Cognitive fatigue in audio work is real, and automation helps maintain decision quality over long sessions.

Focus on Creativity

At the heart of every great mix is a set of creative decisions: how do you want the drums to feel? Should the vocal float above the beat or sit inside it? Is the chorus too loud, or just loud enough to hit emotionally?

These questions require intuition, artistry, and emotional focus, none of which are easy to summon when you're knee-deep in technical tweaks (Moylan, 2014). Workflow automation carves out mental space and session time for this kind of creative flow. By removing the friction of mundane tasks, automation puts the engineer back in touch with the *why* of mixing, not just the *how*.

In fact, many modern mixers rely on highly customized templates with automated gain structure setups, pre-routed buses, and even smart macros to reset plugin chains or prepare parallel processing chains. This allows them to dive directly into the emotional architecture of the song without wasting energy on the plumbing.

For example, when you're working on a moody ballad and you want the reverb tail to swell just after the chorus ends, you're thinking about *feeling*, not *gain math*. Workflow automation gives you more chances to be in that zone; focused, inspired, and emotionally connected to the music.

In the evolving landscape of music production, automation is no longer just about speed, it's about sustainability. Sustainable creativity. Sustainable energy. Sustainable sound quality. Whether you're a solo bedroom producer, a commercial engineer, or a hybrid artist doing everything yourself, automating your workflow can be the difference between burnout and brilliance. It allows you to mix smarter, not harder. And most importantly, it allows the music, not the meters, to be at the center of your process (Izhaki, 2017).

2.3.4 Gaps & Why This Project Is Needed

The systems mentioned above fall into one of two categories: **academic multi-track optimizers** or **manual batch tools and metering utilities**. No open-source or academic project currently offers:

1. **Real-time, plugin-aware gain compensation** capable of dynamically measuring LUFS-I before and after bypass.
2. **Seamless DAW integration**, operating within tracks, buses, and the master, not just isolated clips.
3. **Transparent volume matching** that respects existing gain staging and only adjusts for bypass-induced differences.

My tool bridges these gaps. It places LUFS-based measurement and compensation inside a JSFX plugin, controlled via Lua scripting. This allows for live compensation during plugin bypass ensuring consistent perceived loudness without disrupting mix balance. In effect, my work, to my best knowledge, is the first truly **real-time, DAW-native, LUFS-driven, bypass-aware gain compensation system.**

CHAPTER THREE

SYSTEM ANALYSIS AND DESIGN

3.0 Introduction

This chapter outlines the analysis of existing systems and the design of the proposed "Auto Gain Match" system. Existing audio processing workflows in digital audio workstations (DAWs) often require manual adjustments to match the loudness of a processed audio signal to its original loudness. This is a time-consuming process that can introduce human error and inconsistencies. The proposed system aims to automate this process, providing an efficient and accurate solution for gain matching. It analyzes audio loudness before and after an effect (FX) and automatically applies the necessary gain compensation to the output.

The system analysis section will provide a detailed look at the current manual gain-matching process, identify its problems, and define the requirements for the new system. The design section will then present the architecture and components of the proposed system.

3.1 System Analysis

3.1.1 Overview of Existing Systems

The existing method for gain matching involves inserting a loudness meter after the plugin. The user then plays the audio twice: once with the plugin bypassed and once with it active. During this process, the user mentally notes, though often poorly, the gain difference. After noting the difference, they manually adjust to the perceived gain difference and then delete the analysis plugin. This manual and imprecise process is repeated for each track and FX chain, making it inefficient for large projects.

The main flaw of this existing system is its reliance on manual intervention. This can lead to inaccuracies and is not suitable for a fast-paced production environment.

3.1.2 Problem Identification

Based on the analysis of the existing system, the following problems are identified:

1. **Time Consumption:** Manual gain matching is a slow process that requires multiple steps, including inserting meters, playing back audio, and mentally noting down values.
2. **Inaccuracy:** Human error can occur when reading, writing, or calculating gain differences, leading to inconsistent results.
3. **Lack of Automation:** The process lacks automation, requiring repetitive actions for each track or effect.
4. **Inefficiency:** Manually inserting and removing meters is exhausting and disrupts the user's workflow.
5. **Complexity:** Non-technical users may find it challenging to correctly interpret loudness measurements and apply appropriate gain compensation.

3.1.3 User Requirements

The user requirements for the proposed system are categorized into functional and non-functional requirements.

Functional Requirements

1. The system must analyze a selected audio range to measure both the input and output loudness of an effect plugin.
2. The system must capture loudness data for True Peak and LUFS-Integrated (LUFS-I) values for both input and output signals.
3. The system must automatically calculate the gain difference between the input and output loudness values.

4. The system must provide the user with a choice to apply gain compensation based on either the True Peak or LUFS-I gain difference.
5. The system must automatically insert a gain plugin after the target effect and set its gain to the calculated value.
6. The system must be able to handle both master tracks and individual tracks.
7. The system must automatically remove the analysis meters after the process is complete.

Non-Functional Requirements

1. **Usability:** The system should have a simple and intuitive interface. It should be easy for a user to select an effect and initiate the analysis.
2. **Performance:** The system should perform the analysis and apply the gain faster than manual and efficiently.
3. **Security:** The system operates locally within the DAW environment and does not handle sensitive data, so security is not a primary concern. The codebase is transparent to the user. They can access and manipulate the source code.
4. **Scalability:** The system is scalable as it can be applied to any number of tracks and FX chains within a project.

3.1.4 Feasibility Study

A feasibility study was conducted to determine the viability of the proposed system.

Technical Feasibility

The proposed system is technically feasible as it leverages the existing scripting and plugin architecture of the REAPER digital audio workstation. The system is built using JSFX (JSFX Plugins) for metering and gain control and Lua for scripting the automation logic. This approach requires no new hardware, only the software components available within the DAW environment.

Economic Feasibility

The system is economically feasible because its development cost is minimal. It uses free and open-source languages (JSFX and Lua) and does not require purchasing additional software or hardware. The gain in efficiency and accuracy from automating the gain-matching process outweighs the time invested in its development.

Operational Feasibility

The system is operationally feasible. It is designed to integrate seamlessly into a common audio production workflow. Users who are accustomed to working with effects and scripts within a DAW will find the process intuitive. The automation of a repetitive task will likely be welcomed and readily adopted by users.

3.1.5 Data Collection Analysis

Data for this project would be collected through a review of existing practices and the analysis of the source code to be built.

1. **Review:** A review of the typical manual workflow for gain matching in DAWs provided the foundational understanding of the problem. This helped in identifying the core issues of inefficiency and inaccuracy.
2. **Observation:** The analysis of the JSFX plugins and Lua scripts served as a form of observation. The code revealed the specific methods for measuring loudness, inserting plugins, and manipulating parameters. For example, the `auto_gain_match_meter` JSFX plugin captures and outputs True Peak and LUFS-I values to a shared memory space (`gmem`). The Lua analysis script then reads these values to calculate the gain differences.

Findings

The findings confirm that the system can be built by combining existing technologies. The JSFX plugins provide the core audio processing functionality (metering, gain), while the

Lua scripts act as the "glue" to automate the workflow, interact with the user, and manage the plugins.

3.2 System Design

3.2.1 System Architecture

The proposed system architecture is a three-tier model: Presentation Layer, Application Layer, and Data Layer.

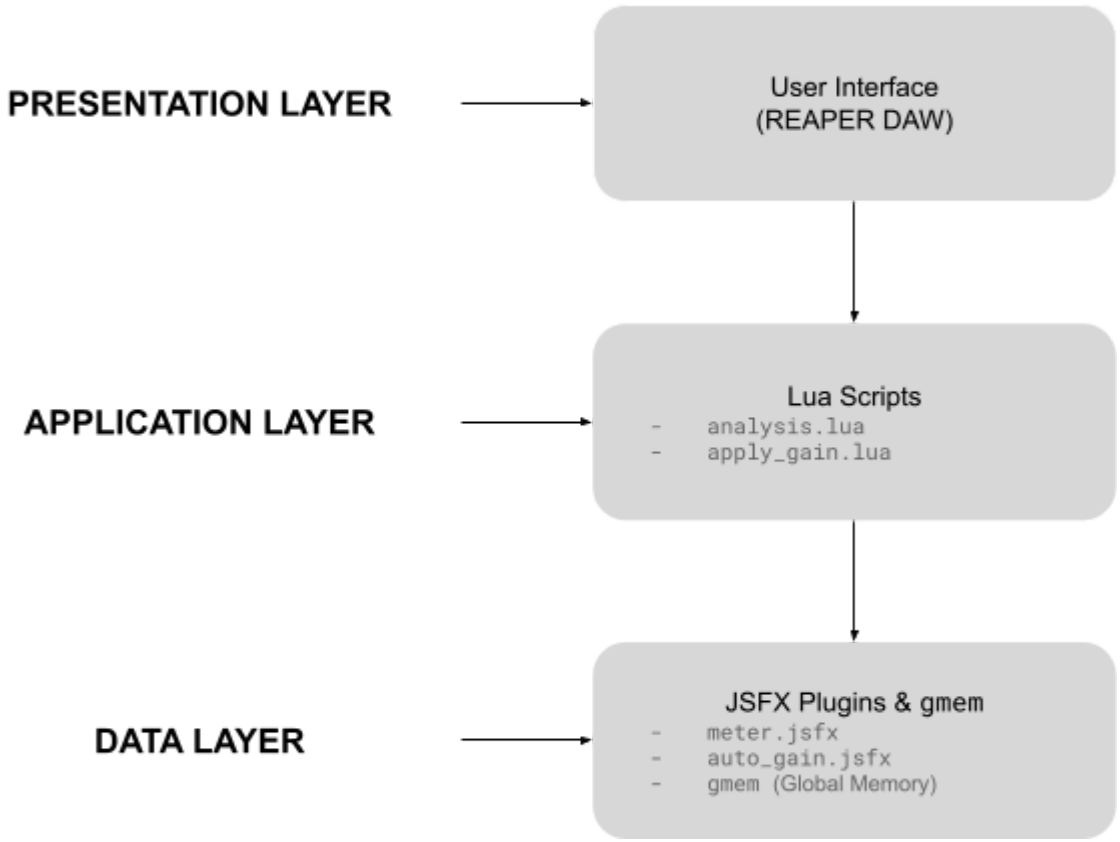

Figure 3.1: The System Architecture of Auto Gain Match System

1. **Presentation Layer:** User Interface (REAPER DAW)

   This layer is the user-facing interface, represented by the REAPER Digital Audio Workstation.

2. **Application Layer:** Lua Scripts

   This is the core "brain" of the system. It consists of two main Lua scripts logic embedded in one (`analysis` and `apply_gain`).

   1. `analysis`: Manages the workflow. It inserts the meters, starts playback, reads data, calculates differences, and presents results to the user.
   2. `apply_gain`: Inserts the gain plugin and applies the calculated gain value.

3. **Data Layer):** JSFX Plugins & `gmem`

   This layer handles the low-level audio processing and data storage. Its subcomponents are:

   1. `auto_gain_match_meter` JSFX: Performs audio analysis (True Peak, LUFS-I) and writes the results to `gmem`.
   2. `Auto Gain` JSFX: Applies the gain to the audio signal.
   3. `gmem` (Global Memory): A shared memory space used by the JSFX plugins to store and retrieve the measured loudness values.

3.2.3 Input Design

The input design consists of a single point of interaction: the user's selection of an effect plugin parameter in the REAPER DAW. The system relies on the `reaper.GetLastTouchedFX()` function to identify the target effect and track. This eliminates the need for a separate input form.

The `analysis` script's first message box can be considered a form of input design, as it prompts the user and confirms the detected FX context.

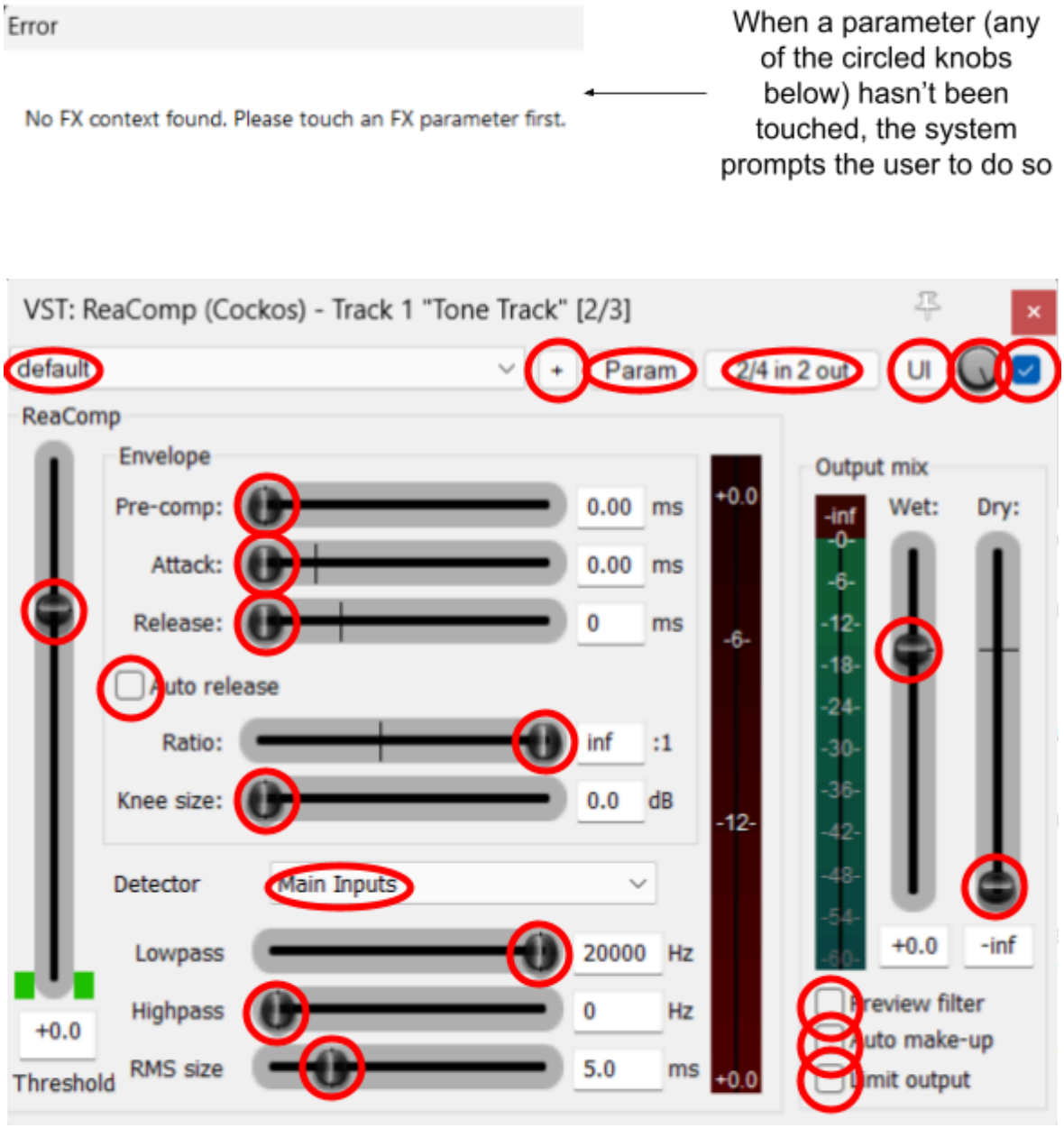

Figure 3.2: Highlighted parameters of a plugin.

3.2.4 Output Design

The system's output is presented to the user through several means:

1. **Loudness Meter Visuals:** The `auto_gain_match_meter` plugin displays real-time loudness values visually with meters and numerical readouts.
2. **Console/Messages Boxes:** The Lua script outputs detailed analysis results to the REAPER console/window, including input/output values and the calculated gain differences. The system also uses the message boxes to confirm operations and present final analysis results, including the option to apply the gain.

3.2.5 Database Design

This system does not require a formal database. Instead, it uses two forms of temporary data storage:

1. **Shared Global Memory (`gmem`):** Used by the JSFX plugins to pass real-time loudness data between the meters and the Lua scripts. This is a simple key-value store (`gmem[0] = in_peak`, `gmem[1] = in_lufs_i`, etc.).
2. **ExtState:** A persistent key-value store within REAPER that the Lua scripts use to save the FX context and analysis results (`input_peak`, `lufs_i_gain`, etc.) between script runs.

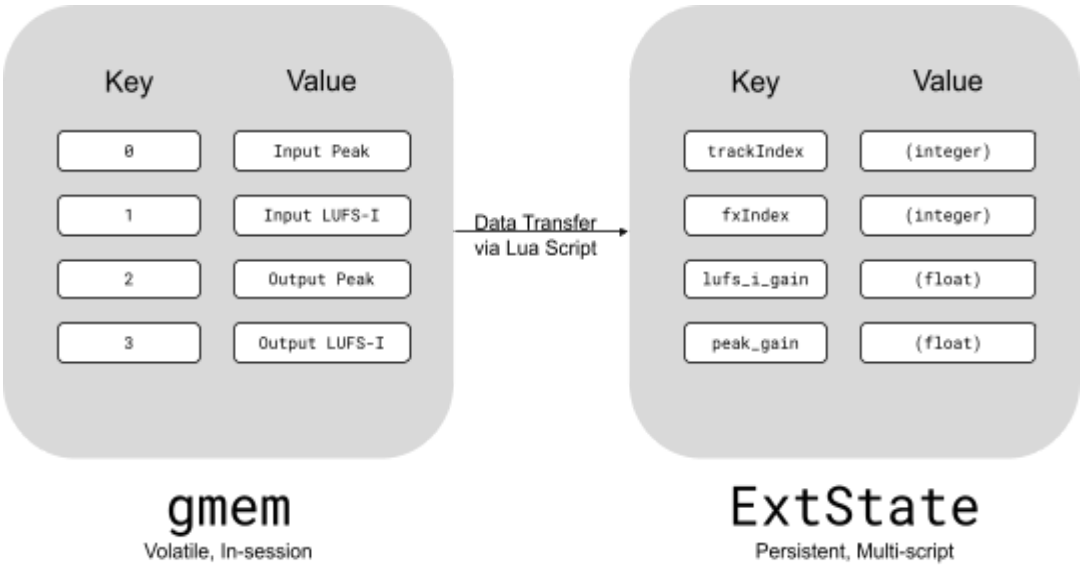

Figure 3.3: The Database Design of Auto Gain Match System

3.2.6 Process Modeling

Data Flow Diagram (DFD)

Figure 3.4: Data Flow Diagram between the User and Software

Use Case Diagram

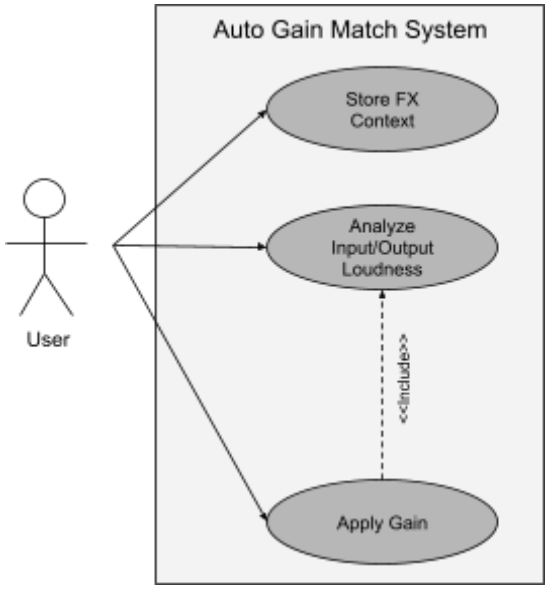

Figure 3.5: Use Case Diagram for the Auto Gain Match Software

Sequence Diagram

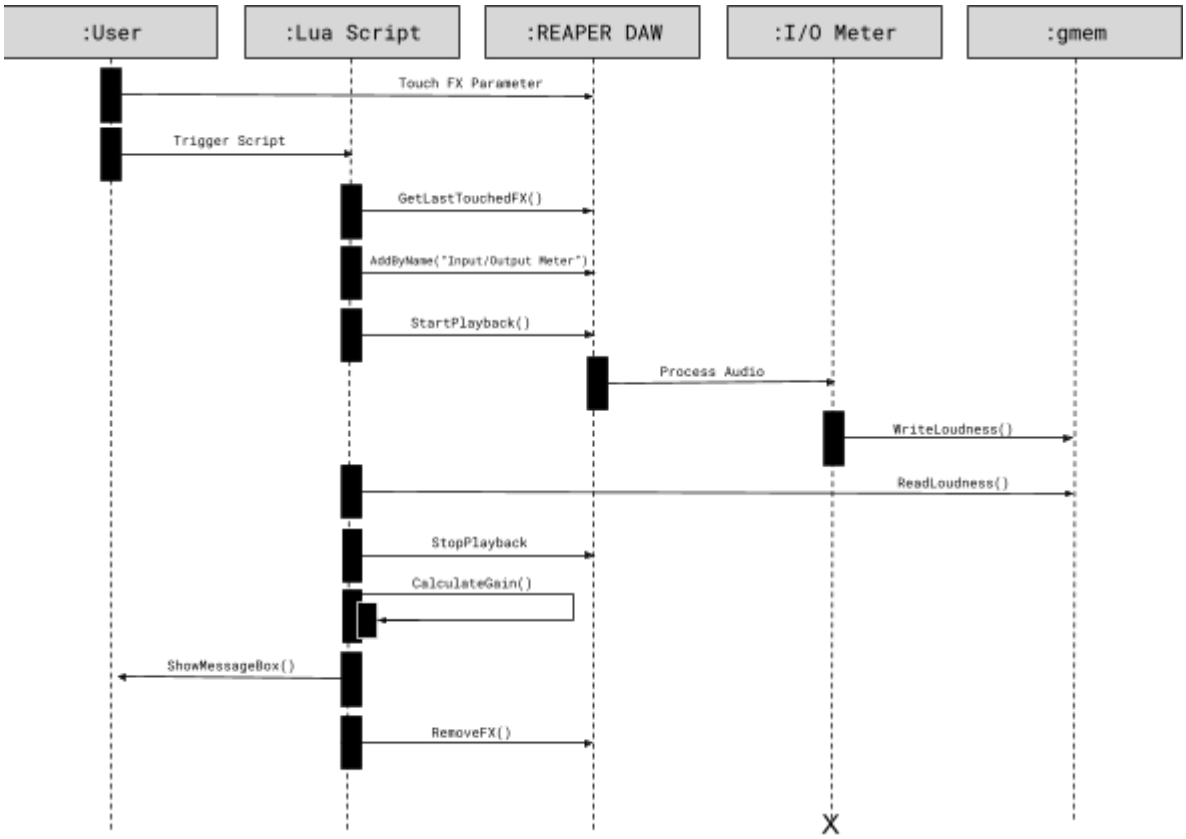

Figure 3.6: Sequence Diagram for the Auto Gain Match Software Elements

CHAPTER FOUR

SYSTEM IMPLEMENTATION

4.0 Introduction

This chapter details the practical implementation of the Auto Gain Match system, building upon the design specifications outlined in the previous chapter. It covers the tools and environments used for development, the phased implementation process including coding, database handling, interface design, and security measures, followed by testing, deployment, and user training. The focus is on how the theoretical designs from Chapter 3 were translated into a functional REAPER script that analyzes audio input/output loudness and applies automated gain adjustments using custom JSFX plugins.

The transition from Chapter 3 to this chapter shifts from conceptual design and architecture to hands-on development and execution. In Chapter 3, we defined the system's requirements, data flows, and high-level algorithms for loudness metering and gain compensation. Here, we describe the actual building, testing, and rollout of the system, ensuring it meets the goals of seamless integration within the REAPER digital audio workstation (DAW) environment for efficient audio processing.

4.1 System Development Tools and Environments

We built the system using a combination of software tools tailored for REAPER scripting and audio plugin development. The primary software tools included:

1. **REAPER DAW (version 7.x or later)**: The core platform for hosting and executing the script, providing built-in APIs for track manipulation, FX insertion, and playback control.

2. **Lua programming language (embedded in REAPER)**: Used for the main script logic, including FX context storage, meter insertion, playback handling, and gain application.
3. **JSFX (REAPER's JavaScript Effects language)**: Employed to create custom plugins for loudness metering ("auto_gain_match_meter.jsfx") and gain adjustment ("Auto Gain.jsfx").
4. **ReaScript editor**: REAPER's IDE For writing and debugging Lua and JSFX code.

Figure 4.1 REAPER's integrated development environment.

5. **REAPER's SWS Extension**: To facilitate script testing, loudness confirmation and console output.

4.1.1 Hardware Requirements

For hardware, we utilized a standard personal computer setup, including:

1. **Processor**: A minimum of a dual-core processor or equivalent.
2. **RAM**: 4 GB minimum, .
3. **Storage**: SSD with at least 1 GB free space.
4. **Audio Interface**: ASIO-compatible sound card for low-latency playback.

4.1.2 Software Requirements

The development environment was set up within the **REAPER DAW** itself. We configured REAPER's Actions List to load and run **Lua scripts**, enabled **JSFX** plugin development via the REAPER effects folder, and used **REAPER's IDE** for real-time testing. Code was version-controlled using **Git** for backups, and debugging was performed via **REAPER's console** and message boxes. The environment allowed for iterative development, where scripts could be reloaded on-the-fly without restarting REAPER.

4.1.3 System Development Environment Architecture

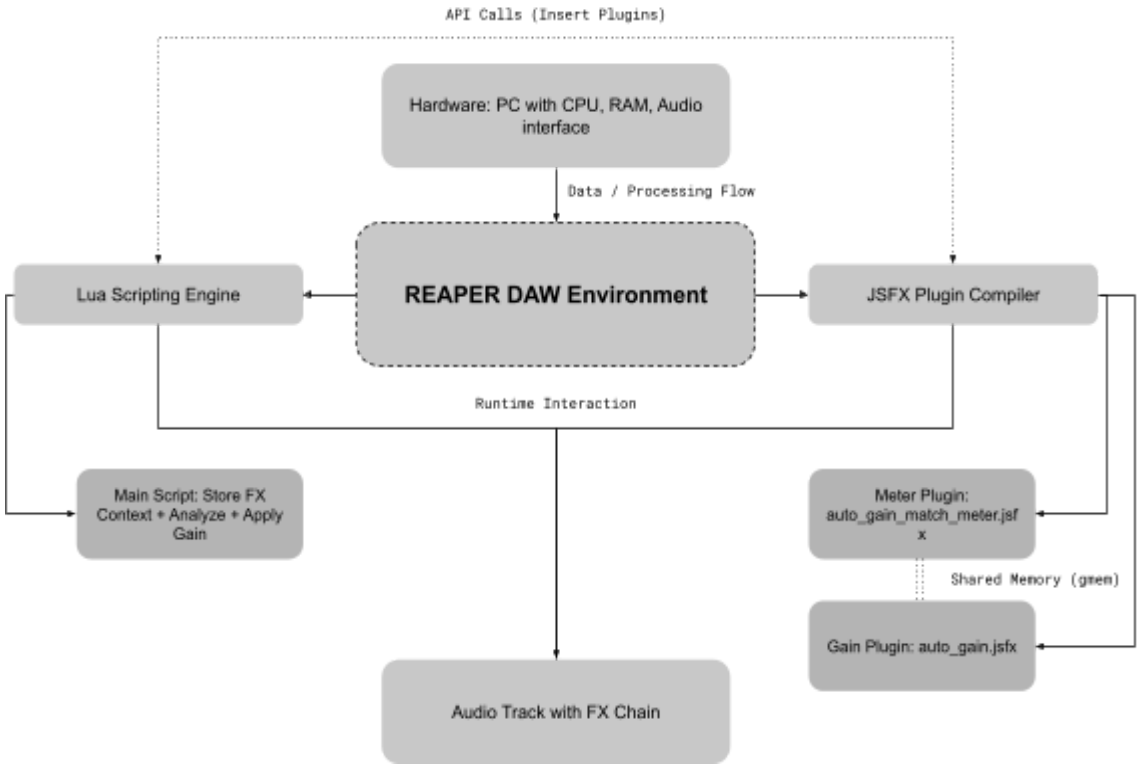

Figure 4.2: The System Development Environment Architecture, showing how the hardware and software interact.

4.2 System Implementation Phases

The implementation was divided into distinct phases:

4.2.1 Coding and Programming

We began with coding the core logic in Lua, focusing on FX context storage, meter insertion, playback control, and gain application. The script handles fresh FX touches via `reaper.GetLastTouchedFX()` or falls back to stored ExtState values. Key functionality includes inserting meters before and after the target FX, playing the audio range, capturing loudness via `gmem`, and applying clamped gain (-15 to +15 dB) to an "Auto Gain" JSFX.

Figure 4.3 highlights crucial code snippets for the key **insertion** logic. For example:

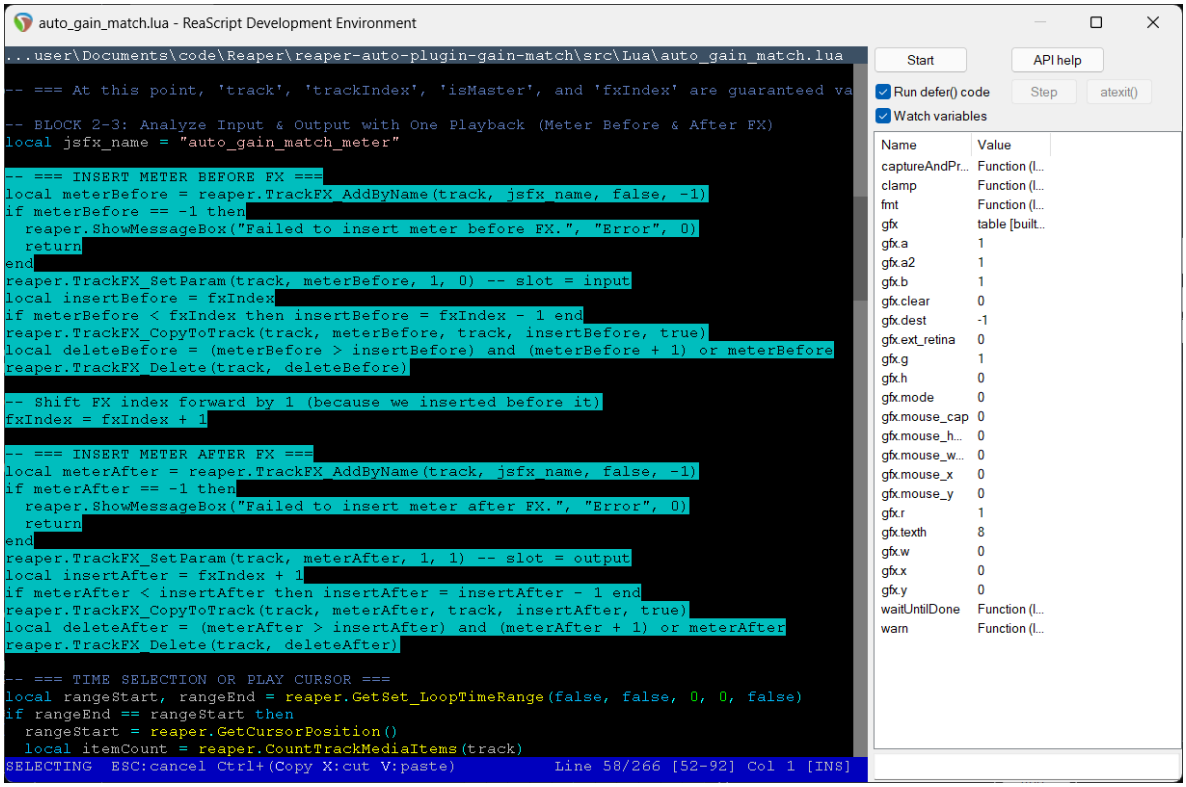

Figure 4.3: a highlighted insertion Lua script in REAPER's code editor

Figure 4.4, another snippet, shows the **capture** logic:

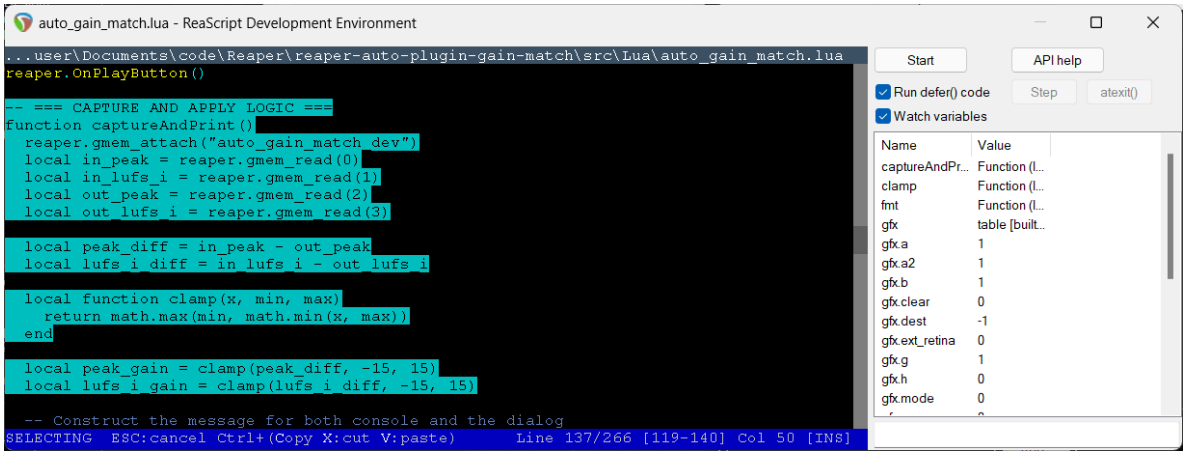

Figure 4.4: a highlighted data capture Lua script in REAPER's code editor

A broader scope of the entire code is highlighted in the appendix

4.2.2 Database Implementation

Although the system does not use a traditional relational database, we implemented data storage using REAPER's `ExtState` for persistent context (e.g., track/FX indices, names) and global memory (`gmem`) for real-time loudness data sharing between JSFX meters. This acts as a lightweight "database" for temporary and session-persistent data.

The process involved creating ExtState keys for storage/retrieval and attaching gmem for inter-plugin communication. Relationships include: ExtState holds static context (one-to-one with FX instances), while gmem slots handle dynamic loudness values (input/output peaks and LUFS-I, mapped as array indices 0-3).

4.2.3 Samples of Queries

Samples of queries that can be run include:

1. **Store**: `reaper.SetExtState("AutoGainMatch", "peak_gain", tostring(peak_gain), false)`
2. **Retrieve**: `local peak_gain = tonumber(reaper.GetExtState("AutoGainMatch", "peak_gain"))`
3. **GMEM read**: `local in_peak = reaper.gmem_read(0)`

4.2.4 Interface Implementation

The client interacts with the system via REAPER's script execution (e.g., from the Actions List) and modal dialogs for results and choices. No custom GUI was built; instead, we leveraged REAPER's message boxes for user input, such as selecting LUFS-I or Peak gain application.

Figures illustrate the UI elements: In Figure 4.5, for the analysis, the input and output meters show a graphical representation of the volume in LUFS-I and Peak while working through to analyse the loudness.

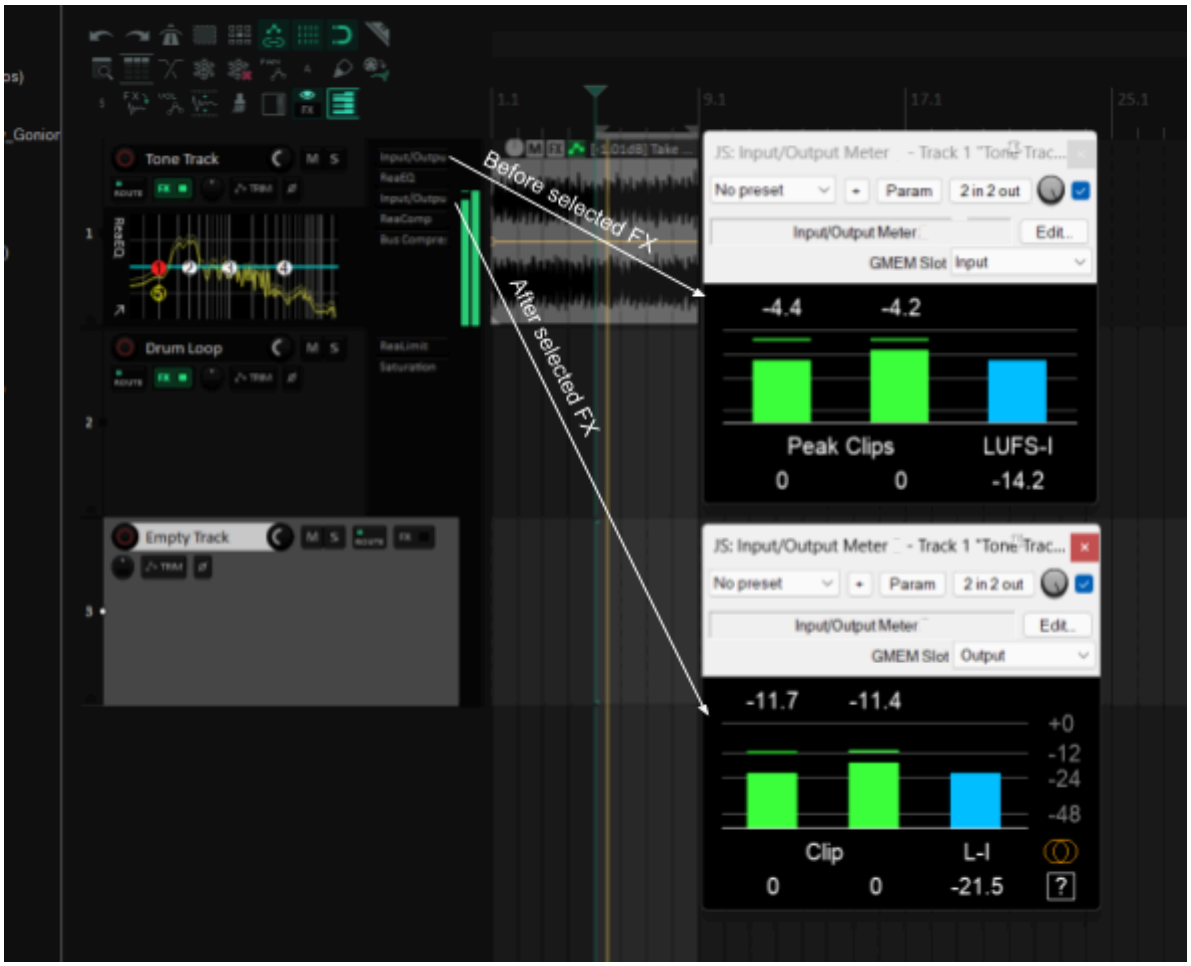

Figure 4.5: a real time snapshot of the script at work, inputting the analysis plugins in the chain

For the results dialog:

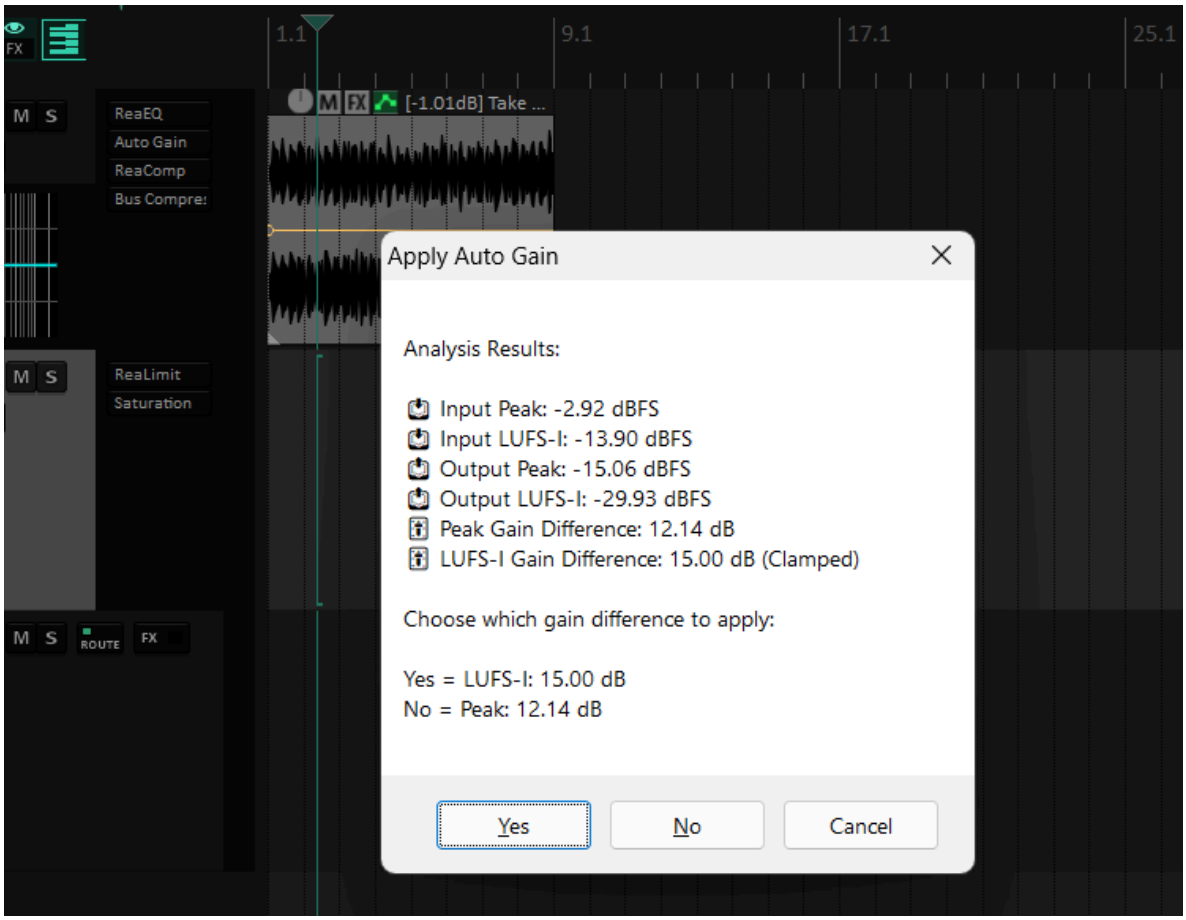

Figure 4.6: A Lua script's dialog displaying the values of input and output gain and their difference.

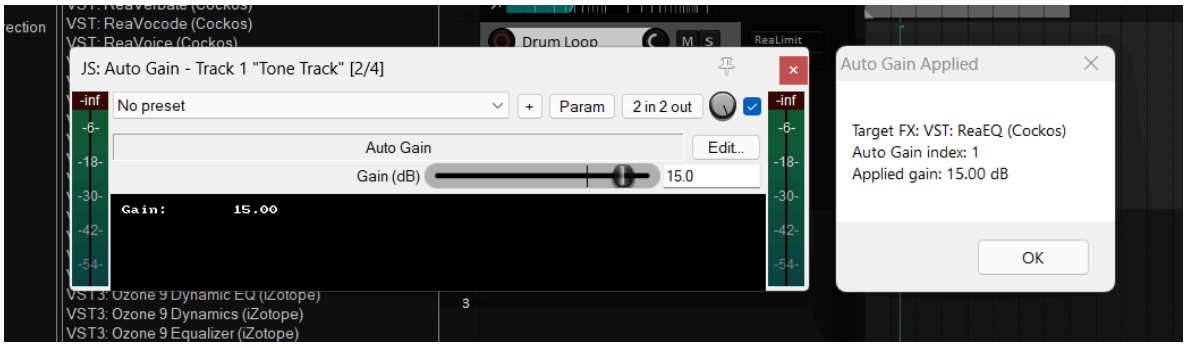

Figure 4.7: A Lua script's dialog for the confirmation and application of gain difference in the Auto Gain plugin.

4.2.5 Security Features

We implemented basic security features focused on data integrity and error handling, such as validating FX indices to prevent crashes and clamping gain values to avoid audio distortion. User authentication is handled implicitly via REAPER's session, with no explicit login required as the system is for local use. Authorization ensures only valid tracks/FX are targeted, with fallbacks for invalid contexts. Figure 4.8 depicts the user authentication process flowchart.

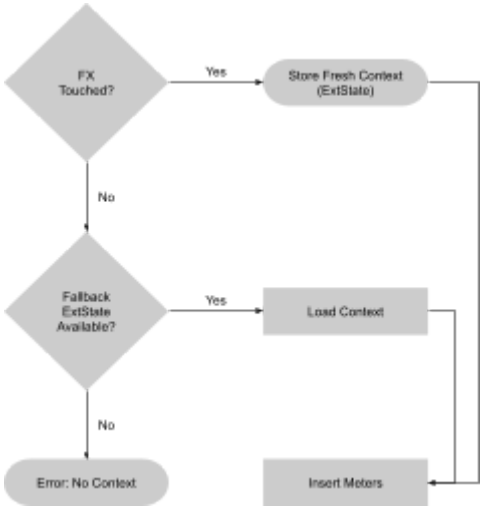

Figure 4.8: Flowchart for User Authentication Process

4.3 System Evaluation and Testing

We employed a combination of unit testing, integration testing, and pilot testing strategies. Unit testing verified individual functions (e.g., meter insertion, `gmem` reads). Integration testing ensured seamless interaction between Lua script and JSFX plugins. Pilot testing involved running the script on sample REAPER projects with varied audio clips to simulate real-world use.

Table 4.1: Sample Test Cases

Test Case ID	Description	Input	Expected Output	Actual Output
TC4.1	Fresh FX touch and analysis	Touch FX param, play audio	Gain diff calculated, dialog shown	Gain diff: 2.5 dB, dialog appears
TC4.2	Fallback context usage	No fresh touch, use ExtState	Loads saved FX, applies gain	Successfully loads, gain: 1.8 dB
TC4.3	Invalid track	Bad track index	Error message	"Invalid track pointer" shown
TC4.4	Clamping extreme gain	Diff: 20 dB	Clamped to 15 dB	Applied: 15 dB
TC4.5	No time selection	Play from cursor	Uses item length	Analyzes full item, correct results

4.4 System Deployment

We used a direct cutover deployment strategy, where the script and JSFX files are installed directly into REAPER, replacing any prior versions instantly upon reload.

Installation steps:

1. **Server (N/A, local system)**: Copy Lua script to REAPER's Scripts folder, JSFX to Effects folder.
2. **End-user's computer**: Open REAPER, go to Options > Show REAPER resource path, place files in appropriate subfolders, then reload actions via Actions > Show action list > Rebuild.

4.5 User Training and Documentation

We created a manual guide for both the user and administrator. The user guide includes steps to run the script (e.g., assign to shortcut, interpret results), while the administrator guide covers code modifications and JSFX debugging.

Training was conducted via online tutorials and hands-on sessions for stakeholders, such as audio engineers. Optional tips and tricks for faster workflow were included for power users.

CHAPTER FIVE

SUMMARY, CONCLUSION AND RECOMMENDATIONS

5.1 Summary

This final year project addressed a critical operational challenge in modern digital audio production: the issue of inconsistent perceived loudness (gain staging) when bypassing or engaging audio effects. Traditional gain staging is a manual, repetitive, and often inaccurate process that leads to unreliable A/B comparisons and increased cognitive load for the audio engineer.

We successfully developed and implemented an Auto Gain Match system, a real-time, automated loudness compensation system directly within the REAPER Digital Audio Workstation (DAW) environment. The solution was built using Lua ReaScripts for control logic and custom JSFX (Jesusonic Effects) for high-precision audio analysis, leveraging the LUFS-I (Integrated Loudness Units Full Scale) standard for accurate, relevant measurement that mimics the natural human hearing sensitivity.

The completed system functions by continuously measuring the loudness of an audio track before and after an insert effect. The Lua script then calculates the required gain difference and automatically inserts a compensating gain utility to maintain a consistent volume level when the effect is engaged or bypassed.

Chapter 1 introduced the problem and defined the objectives to build a native, bypass-aware compensation system. Chapter 2 reviewed existing literature on loudness standards, mixing practice, and the impact of cognitive fatigue, establishing the academic and practical need for the system. Chapter 3 detailed the system analysis, defined the functional and non-functional requirements, and designed the three-tier system architecture utilizing REAPER's native tools (Lua, JSFX, and ExtState/gmem). Chapter 4 covered the

implementation of the core logic, explaining the use of ExtState for data persistence and gmem for real-time communication between the metering JSFX and the master Lua script.

The project delivered a fully functional, DAW-native solution that effectively automates the process of gain matching, successfully mitigating a significant workflow impediment and ensuring accurate, loudness-consistent A/B testing for audio professionals.

5.2 Conclusion

The Auto Gain Match project successfully addressed the primary objective of mitigating the critical workflow flaw associated with inconsistent and manual gain staging during the A/B comparison of audio effects.

Through systematic literature review, robust system design (Chapter 3), and rigorous implementation using Lua ReaScripts and JSFX (Chapter 4), the project delivered a fully functional, DAW-native solution. The system successfully monitors the Integrated Loudness (LUFS-I) of a signal and automatically applies precise gain compensation when effects are engaged or bypassed.

The project validates the feasibility of using deep, native DAW scripting to solve complex, real-world problems in audio engineering. By automating this crucial technical task, Auto Gain Match dramatically improves the accuracy of subjective listening tests, reduces the cognitive load on the engineer, and promotes a more objective and efficient workflow. Ultimately, the system provides a valuable tool that allows audio professionals to make more informed creative decisions, confident that they are comparing the *sound* of the plugin, not merely its *volume*.

5.3 Recommendations

While the Auto Gain Match system meets its core objectives, future development should focus on enhancing speed and broader utility. The following recommendations are proposed:

1. **Non-Real-Time Measurement (Offline Analysis):** To eliminate the workflow disruption caused by lengthy real-time LUFS measurement, the system must implement a faster-than-real-time (offline) analysis mode. This would involve scripting a process to rapidly read the selected audio waveform and instantly calculate the required gain correction, significantly reducing the wait time needed for playback and measurement.
2. **Expanded Data Persistence:** Implement a dedicated storage mechanism (beyond `ExtState`) to manage and instantly recall specific gain settings for every plugin instance across a project, further minimizing recalculation time when loading sessions.
3. **Cross-Platform Portability:** Explore the portability of the core calculation logic to other major DAWs (e.g., via Python or other APIs) to maximize the system's impact across the professional audio industry.
4. **Adaptive Loudness Targeting:** Introduce a dynamic component to the system that ensures the final compensated loudness adheres to common delivery standards (e.g., -14 LUFS for streaming), making it suitable for both gain staging and final-mix compliance.

REFERENCES

EBU. (2011). *EBU R 128: Loudness normalisation and permitted maximum level of audio signals.* European Broadcasting Union.

EBU. (2014). *Tech 3343: Loudness metering - Streaming*. European Broadcasting Union.

Giannoulis, D., Massberg, M., & Reiss, J. D. (2012). Dynamic range compression: History, applications and standards. *Journal of the Audio Engineering Society*, 60(4), 249–271.

Huber, D. M., & Runstein, R. E. (2017). *Modern recording techniques* (9th ed.). Focal Press.

Izhaki, R. (2017). *Mixing audio: Concepts, practices, and tools* (2nd ed.). Routledge.

Katz, B. (2015). *Mastering audio: The art and the science* (3rd ed.). Focal Press.

Moylan, W. (2014). *Understanding and crafting the mix: The art of recording*. Focal Press.

Senior, M. (2011). *Mixing secrets for the small studio*. Focal Press.

ITU. (2015). *Recommendation BS.1770-4: Algorithms to measure audio programme loudness and true-peak audio level*. International Telecommunication Union.

Vickers, E. (2010). The loudness war: Is there an end in sight? *AES Convention Paper*.

Ward, S., Gonzalez, L., & Reiss, J. D. (2023). Modeling partial loudness for automated mix balancing. *Journal of the Audio Engineering Society*, 71(4), 170–185.

Zölzer, U. (2011). *DAFX: Digital audio effects*. Wiley.

Appendix A

A Code Snippet On Storing Fx And Track Context In Lua Script

```
local retval, fakeTrack, fxIndex, paramIndex = reaper.GetLastTouchedFX()
local trackIndex, isMaster, track

if retval then
  trackIndex = tonumber(fakeTrack)
  isMaster = (trackIndex == 0)
  track = reaper.CSurf_TrackFromID(trackIndex, isMaster)
      local trackName = isMaster and "[MASTER TRACK]" or (select(2,
reaper.GetTrackName(track, "")) or "[unnamed]")
  local _, fxName = reaper.TrackFX_GetFXName(track, fxIndex, "")
   local _, paramName = reaper.TrackFX_GetParamName(track, fxIndex, paramIndex,
"")
  local value = reaper.TrackFX_GetParam(track, fxIndex, paramIndex)
  reaper.SetExtState("AutoGainMatch", "trackIndex", tostring(trackIndex), false)
  reaper.SetExtState("AutoGainMatch", "isMaster", tostring(isMaster), false)
  reaper.SetExtState("AutoGainMatch", "fxIndex", tostring(fxIndex), false)
  reaper.SetExtState("AutoGainMatch", "paramIndex", tostring(paramIndex), false)
  reaper.SetExtState("AutoGainMatch", "value", tostring(value), false)
  reaper.SetExtState("AutoGainMatch", "trackName", trackName, false)
  reaper.SetExtState("AutoGainMatch", "fxName", fxName, false)
  reaper.SetExtState("AutoGainMatch", "paramName", paramName, false)
else
  trackIndex = tonumber(reaper.GetExtState("AutoGainMatch", "trackIndex") or "")
  isMaster = reaper.GetExtState("AutoGainMatch", "isMaster") == "true"
  fxIndex = tonumber(reaper.GetExtState("AutoGainMatch", "fxIndex") or "")
  if not trackIndex or not fxIndex then
      reaper.ShowMessageBox("No FX context found. Please touch an FX parameter
first.", "Error", 0)
    return
  end
  track = reaper.CSurf_TrackFromID(trackIndex, isMaster)
```

```
  paramIndex = tonumber(reaper.GetExtState("AutoGainMatch", "paramIndex") or "")
end
```

Appendix B

Inserting Meters Before/After Fx In Lua Script For Reaper

```
local jsfx_name = "auto_gain_match_meter"
local meterBefore = reaper.TrackFX_AddByName(track, jsfx_name, false, -1)
reaper.TrackFX_SetParam(track, meterBefore, 1, 0) -- slot = input
local insertBefore = fxIndex
if meterBefore < fxIndex then insertBefore = fxIndex - 1 end
reaper.TrackFX_CopyToTrack(track, meterBefore, track, insertBefore, true)
local deleteBefore = (meterBefore > insertBefore) and (meterBefore + 1) or
meterBefore
reaper.TrackFX_Delete(track, deleteBefore)
fxIndex = fxIndex + 1  -- Shift FX index

local meterAfter = reaper.TrackFX_AddByName(track, jsfx_name, false, -1)
reaper.TrackFX_SetParam(track, meterAfter, 1, 1) -- slot = output
local insertAfter = fxIndex + 1
if meterAfter < insertAfter then insertAfter = insertAfter - 1 end
reaper.TrackFX_CopyToTrack(track, meterAfter, track, insertAfter, true)
local deleteAfter = (meterAfter > insertAfter) and (meterAfter + 1) or meterAfter
reaper.TrackFX_Delete(track, deleteAfter)
```

Appendix C

Playback Control, Time Selection And Application Of Gain In Luascript

```
local rangeStart, rangeEnd = reaper.GetSet_LoopTimeRange(false, false, 0, 0,
false)
if rangeEnd == rangeStart then
  rangeStart = reaper.GetCursorPosition()
  local itemCount = reaper.CountTrackMediaItems(track)
  rangeEnd = -math.huge
  for i = 0, itemCount - 1 do
    local item = reaper.GetTrackMediaItem(track, i)
    local pos = reaper.GetMediaItemInfo_Value(item, "D_POSITION")
    local len = reaper.GetMediaItemInfo_Value(item, "D_LENGTH")
    local itemEnd = pos + len
    if itemEnd > rangeEnd then rangeEnd = itemEnd end
  end
end

local loopCommand = 1068
local wasLooping = reaper.GetToggleCommandState(loopCommand)
if wasLooping == 1 then reaper.Main_OnCommand(loopCommand, 0) end

reaper.SetEditCurPos(rangeStart, true, false)
reaper.OnPlayButton()

local choice = reaper.ShowMessageBox(
    string.format("Analysis Results:\n\n%s\nChoose which gain difference to
apply:\n\nYes = LUFS-I: %.2f dB\nNo = Peak: %.2f dB", results_msg, lufs_i_gain,
peak_gain),
  "Apply Auto Gain",
  3
)

local gainDiff = nil
```

```
if choice == 6 then gainDiff = lufs_i_gain
elseif choice == 7 then gainDiff = peak_gain
else return end

local AUTO_GAIN_FXNAME = "JS: Auto Gain"
local  autoGainIndex  =  reaper.TrackFX_AddByName(track,  AUTO_GAIN_FXNAME,  false,
-1000 - (fxIndex + 1))
if autoGainIndex ~= fxIndex + 1 then
  reaper.TrackFX_CopyToTrack(track, autoGainIndex, track, fxIndex + 1, true)
   local deleteIndex = (autoGainIndex > fxIndex + 1) and (autoGainIndex + 1) or
autoGainIndex
  reaper.TrackFX_Delete(track, deleteIndex)
  autoGainIndex = fxIndex + 1
end

reaper.TrackFX_SetParam(track, autoGainIndex, 0, gainDiff)
```

Appendix D

Meter Initialization And Sample Processing In Jsfx

```
function init_lufs_filters() {
  // High-shelf filter coefficients
  db=3.999843853973347;
  f0=1681.974450955533;
  Q=0.7071752369554196;
  K=tan($pi*f0/srate);
  Vh=pow(10, db/20);
  Vb=pow(Vh, 0.4996667741545416);
  a0=1+K/Q+K*K;
  f1a1=2*(K*K-1)/a0;
  f1a2=(1-K/Q+K*K)/a0;
  f1b0=(Vh+Vb*K/Q+K*K)/a0;
  f1b1=2*(K*K-Vh)/a0;
  f1b2=(Vh-Vb*K/Q+K*K)/a0;

  // Pre-filter coefficients
  f0=38.13547087602444;
  Q=0.5003270373238773;
  K=tan($pi*f0/srate);
  f2a1=2*(K*K-1)/(1+K/Q+K*K);
  f2a2=(1-K/Q+K*K)/(1+K/Q+K*K);
  f2b0=1;
  f2b1=-2;
  f2b2=1;
}

function Reset() {
  init_lufs_filters();
  pk=alloc(nch);
  hipk=alloc(nch);
  clip_cnt=alloc(nch);
```

```
  sinc.sinc_init();
  m_win_cnt=0.4/LOUD_METER_UPDATE;
  win_pos=0;
  win_cnt=0;
  win_len=(LOUD_METER_UPDATE*srate)|0;
  i_win_len=1/(m_win_cnt*win_len);
  lufs_m_sum=0;
  lufs_m_db=-100;
  lufs_a_sum=0;
  lufs_a_sum_cnt=0;
  lufs_i_db = -100;
  lufs_buf=alloc(m_win_cnt);
  lufs_a_hist=alloc(2*NUM_BINS);
  global_peak = 0;
  do_all_channels("ch%d.init(%d); ");
}

Reset();

lval=0;
do_all_channels("ch%d.proc(spl%d); ");

(win_pos += 1) >= win_len ? (
  win_pos=0;
  win_cnt += 1;
  prev_lval=lufs_buf[cur_buf];
  lufs_buf[cur_buf]=lval;
  (cur_buf += 1) >= m_win_cnt ? cur_buf=0;
  lufs_m_sum += (lval-prev_lval)*i_win_len;
  lufs_m_sum > 0 && win_cnt >= m_win_cnt ? (
    lufs_m_db=-0.691+log(lufs_m_sum)*10/log(10);
     a = WANT_INTEGRATED_ALWAYS || (play_state&1) ? ((lufs_m_db+70)*BINS_PER_DB)|0
: -1;
    a >= 0 ? (
      a >= NUM_BINS ? a=NUM_BINS-1;
```

```
      lufs_a_sum += lufs_m_sum;
      lufs_a_sum_cnt += 1;
      lufs_a_hist[2*a] += 1;
      lufs_a_hist[2*a+1] += lufs_m_sum;
      lufs_a_db=-0.691+log(lufs_a_sum/lufs_a_sum_cnt)*10/log(10);
      lufs_a_gate=((lufs_a_db-10+70)*BINS_PER_DB)|0;
      lufs_i_sum=0;
      lufs_i_cnt=0;
      bin=max(lufs_a_gate,0);
      loop(NUM_BINS-bin,
        lufs_i_cnt += lufs_a_hist[2*bin];
        lufs_i_sum += lufs_a_hist[2*bin+1];
        bin += 1;
      );
       lufs_i_db=lufs_i_sum > 0 ? -0.691+log(lufs_i_sum/lufs_i_cnt)*10/log(10) :
-100;
    );
  ) : (
    lufs_m_db=-100;
  );
);

base = slot * 2;
gmem[base] = global_peak > 0 ? log(global_peak)*20/log(10) : -150;
gmem[base+1] = lufs_i_db > -100 ? lufs_i_db : lufs_m_db;
```

Appendix G

Gain Application (@slider and @sample in JSFX Auto Gain Plugin)

```
@slider
nxt_gain = 10 ^ (gain_db / 20);

@block
avg_gain = (nxt_gain - lst_gain) / samplesblock;

@sample
spl0 *= lst_gain;
spl1 *= lst_gain;
lst_gain += avg_gain;
```