



**CHALMERS**  
UNIVERSITY OF TECHNOLOGY



# **Study of Real Time Aspects for Motion-Driven Music Generation for a Quadraped Robot**

Master's thesis in Embedded Electronic System Design

**SHIMIN HU**  
**XIN YI**

Department of Microtechnology and Nanoscience  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Gothenburg, Sweden 2026



MASTER'S THESIS 2026

# Study of Real Time Aspects for Motion-Driven Music Generation for a Quadruped Robot

Shimin Hu  
Xin Yi



Department of Microtechnology and Nanoscience  
CHALMERS UNIVERSITY OF TECHNOLOGY  
Gothenburg, Sweden 2026

Study of Real Time Aspects for Motion-Driven Music Generation for a Quadraped Robot

© SHIMIN HU, 2026.

© XIN YI, 2026.

Supervisor: Shivesh Kumar, Mechanics and Maritime Sciences

Examiner: Per Larsson-Edefors, Microtechnology and Nanoscience

Master's Thesis 2026

Department of Microtechnology and Nanoscience

Chalmers University of Technology

SE-412 96 Gothenburg

Telephone +46 31 772 1000

Typeset in L<sup>A</sup>T<sub>E</sub>X

Gothenburg, Sweden 2026

# Study of Real Time Aspects for Motion-Driven Music Generation for a Quadruped Robot

SHIMIN HU

XIN YI

Department of Microtechnology and Nanoscience  
Chalmers University of Technology

## Abstract

Recent advances in legged robotics have expanded their deployment in human-centered environments, where rich multi-modal interaction is essential. While visual and kinematic behaviors are widely studied, real-time auditory feedback directly driven by robot motion remains comparatively underexplored. Such motion-driven sonification requires bounded low-latency synchronization between physical gait events and audio generation; however, standard general-purpose operating systems and middleware such as ROS 2 may introduce non-deterministic scheduling jitter, delayed callbacks, and audible timing artifacts under computational load.

This thesis proposes and evaluates an embedded audio-motion synchronization framework for the Unitree Go2 quadruped robot. The system captures high-frequency proprioceptive sensor data at 500 Hz, applies lightweight filtering for gait-event extraction, and maps detected motion features to musical parameters and triggered audio responses in real time. To improve temporal predictability, the software architecture combines a decoupled C++ sensing/audio pipeline with operating-system-level real-time scheduling using a `PREEMPT_RT` patched Linux kernel and `SCHED_FIFO` thread prioritization.

Experimental evaluation under simulated autonomous workloads, including SLAM- and navigation-like CPU stress, shows that real-time scheduling primarily improves worst-case timing behavior rather than average computational speed. Under stress, the proposed real-time configuration reduces sensor data loss from 30.16% to 8.30%. It also substantially mitigates long-tail callback execution jitter: while the standard Linux scheduler exhibits a worst-case execution inflation of  $1451\times$  with a maximum callback time of  $2,875\ \mu\text{s}$ , the real-time configuration bounds the corresponding maximum to  $24.7\ \mu\text{s}$ , a  $116\times$  reduction. On the deployed Jetson platform, immediate footstep-to-audio events remain below 40 ms, well within the perceptual synchrony target used in this work. These results demonstrate that careful architectural decoupling and real-time scheduling can make motion-synchronous audio feedback practical for embedded interactive robotic systems.

Keywords: Robotics, Sonification, ROS 2, Real-Time Operating Systems (RTOS), `PREEMPT_RT`, Human-Robot Interaction (HRI), Quadruped Locomotion.



# Acknowledgements

We would like to express our deepest gratitude to everyone who supported us throughout the duration of this master’s thesis. This project would not have been possible without the guidance, resources, and encouragement we received.

First and foremost, we are immensely grateful to our supervisor, Shivesh Kumar, for introducing us to this fascinating topic and for his unwavering guidance. Thank you for the weekly meetings that kept our progress on track, for consistently pointing us in the right research direction, and for providing the essential hardware resources required to bring this project to life. We also deeply appreciate your efforts in facilitating cross-team collaborations, ensuring that our work could seamlessly integrate with and benefit from the resources of other project groups within the lab. Your meticulous feedback on our thesis manuscript was invaluable.

We would also like to extend our sincere thanks to our examiner, Professor Per Larsson-Edefors. Thank you for your early validation and approval of our thesis topic, and for your prompt and helpful consultations whenever we had questions regarding the academic requirements and standards of the master’s thesis process. Furthermore, we wish to thank all the colleagues and fellow students at the Robot Athletic Intelligence Lab (RAIL). A special note of gratitude goes to Xiaoyang Chen for his technical assistance, insightful discussions, and camaraderie during our time in the lab. Working alongside such a supportive and inspiring group of peers has been a truly rewarding experience.

Finally, we thank our families and friends for their endless support and understanding throughout our academic journey.

Shimin Hu & Xin Yi  
Gothenburg, September 2026



# Contents

|          |                                                          |           |
|----------|----------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                      | <b>1</b>  |
| 1.1      | Related Work . . . . .                                   | 1         |
| 1.2      | Purpose and Goal . . . . .                               | 2         |
| 1.3      | Thesis Outline . . . . .                                 | 3         |
| <b>2</b> | <b>Technical Background</b>                              | <b>5</b>  |
| 2.1      | Unitree Go2 Robot Dog . . . . .                          | 5         |
| 2.2      | Quadrupedal Locomotion and Proprioceptive Data . . . . . | 6         |
| 2.2.1    | Gait Cycle and Phases . . . . .                          | 7         |
| 2.2.2    | Proprioceptive Sensors and Data Sources . . . . .        | 7         |
| 2.3      | Signal Processing . . . . .                              | 8         |
| 2.3.1    | Butterworth Low-Pass Filter Theory . . . . .             | 8         |
| 2.3.2    | Filter Group Delay and Transient Response . . . . .      | 9         |
| 2.4      | Robot Operating System 2 (ROS 2) . . . . .               | 10        |
| 2.4.1    | DDS-based Communication Model . . . . .                  | 11        |
| 2.4.2    | Execution Model and Executors . . . . .                  | 11        |
| 2.4.3    | Real-Time Challenges in ROS 2 . . . . .                  | 11        |
| 2.5      | Procedural Audio and Sonification . . . . .              | 12        |
| 2.5.1    | Procedural Audio Synthesis and Sonification . . . . .    | 12        |
| 2.5.2    | Fundamentals of Musical Parameters . . . . .             | 12        |
| 2.5.3    | Amplitude Envelopes (ADSR) . . . . .                     | 13        |
| 2.6      | Real-Time Operating Systems (RTOS) Concepts . . . . .    | 13        |
| 2.6.1    | Hard vs. Soft Real-Time Systems . . . . .                | 14        |
| 2.6.2    | Audio Buffering and Kernel-Level Scheduling . . . . .    | 14        |
| <b>3</b> | <b>Method</b>                                            | <b>17</b> |
| 3.1      | System Overview . . . . .                                | 17        |
| 3.2      | Sonification System . . . . .                            | 18        |
| 3.2.1    | Signal Filtering . . . . .                               | 18        |
| 3.2.2    | Motion-to-Sound Mapping and Music Generation . . . . .   | 18        |
| 3.3      | Real-Time System Design . . . . .                        | 19        |
| 3.4      | Experimental Setup and Hardware Constraints . . . . .    | 19        |
| 3.5      | Metrics for Real-Time Performance . . . . .              | 19        |
| <b>4</b> | <b>Design</b>                                            | <b>21</b> |
| 4.1      | Data Analysis and Motion Extraction . . . . .            | 21        |

|          |                                                               |           |
|----------|---------------------------------------------------------------|-----------|
| 4.1.1    | Robot Original Data . . . . .                                 | 21        |
| 4.1.2    | Butterworth Filter Cutoff Frequency . . . . .                 | 23        |
| 4.1.3    | Butterworth Filter Group Delay . . . . .                      | 24        |
| 4.1.4    | Butterworth Filter Transient Response Analysis . . . . .      | 25        |
| 4.2      | Software Evolution: From Python to C++ . . . . .              | 26        |
| 4.2.1    | Prototyping Phase (Python) . . . . .                          | 26        |
| 4.2.2    | Production Phase (C++) . . . . .                              | 26        |
| 4.3      | Offline Mode . . . . .                                        | 27        |
| 4.3.1    | Rationale for the Offline Baseline . . . . .                  | 27        |
| 4.3.2    | Offline Mode Workflow . . . . .                               | 27        |
| 4.3.3    | Support for Multiple Dance Modes . . . . .                    | 30        |
| 4.4      | Online Mode . . . . .                                         | 30        |
| 4.4.1    | Real-Time Signal Processing Simplification . . . . .          | 30        |
| 4.4.2    | Online Motion-to-Music Algorithm . . . . .                    | 30        |
| 4.5      | Real-Time Integration Strategy . . . . .                      | 33        |
| 4.5.1    | Thread-Safe Communication Between Sensing and Audio . . . . . | 33        |
| 4.5.2    | OS-Level Thread Prioritization . . . . .                      | 33        |
| <b>5</b> | <b>Results</b>                                                | <b>35</b> |
| 5.1      | Offline Dance Mode Results . . . . .                          | 35        |
| 5.1.1    | Dance Motion Snapshot and Feature Mapping . . . . .           | 35        |
| 5.1.2    | Butterworth Low-Pass Filter Performance . . . . .             | 36        |
| 5.1.3    | Motion Extraction from Motor Data . . . . .                   | 36        |
| 5.1.4    | Motion Extraction from IMU Data . . . . .                     | 37        |
| 5.1.5    | Offline Rendering Outcome . . . . .                           | 38        |
| 5.2      | Online Walking Mode Results . . . . .                         | 38        |
| 5.2.1    | Walking Gait Snapshot and Event Detection . . . . .           | 39        |
| 5.2.2    | Behavior across Walking Conditions . . . . .                  | 39        |
| 5.3      | Controlled Real-Time Performance Evaluation . . . . .         | 40        |
| 5.3.1    | Evaluation Platform and Experimental Setup . . . . .          | 41        |
| 5.3.2    | ROS 2 Communication Reliability . . . . .                     | 41        |
| 5.3.3    | Execution Jitter and Priority Inversion . . . . .             | 42        |
| 5.3.4    | End-to-End Latency and Musical Quantization . . . . .         | 44        |
| 5.3.5    | Run-Level Synthesis and Overall Determinism . . . . .         | 44        |
| 5.4      | Onboard Real-World Deployment (Unitree Go2 Jetson) . . . . .  | 46        |
| 5.4.1    | Onboard Jetson Log Dataset . . . . .                          | 47        |
| 5.4.2    | Latency Definitions and Algorithmic Separation . . . . .      | 47        |
| 5.4.3    | Onboard Latency Results . . . . .                             | 49        |
| 5.4.4    | Timing Jitter During On-Robot Operation . . . . .             | 51        |
| 5.4.5    | Online Behavior under Free Walk and Trot Conditions . . . . . | 51        |
| <b>6</b> | <b>Discussion</b>                                             | <b>57</b> |
| 6.1      | Platform Evaluation Scope . . . . .                           | 57        |
| 6.2      | Motion Coverage . . . . .                                     | 58        |
| 6.3      | Role and Limitations of the Offline Mode . . . . .            | 58        |
| 6.4      | Real-Time Architecture and Scheduling . . . . .               | 59        |
| 6.5      | Musical Quality . . . . .                                     | 59        |

|          |                                                              |           |
|----------|--------------------------------------------------------------|-----------|
| 6.6      | Broader Implications for Real-Time Robotic Systems . . . . . | 60        |
| 6.7      | Ethical, Societal and Ecological Aspects . . . . .           | 61        |
| 6.7.1    | Ethical Aspects . . . . .                                    | 61        |
| 6.7.2    | Use of Generative AI . . . . .                               | 61        |
| 6.7.3    | Societal Aspects . . . . .                                   | 62        |
| 6.7.4    | Ecological Aspects . . . . .                                 | 62        |
| <b>7</b> | <b>Conclusion</b>                                            | <b>63</b> |
|          | <b>Bibliography</b>                                          | <b>65</b> |



# 1

## Introduction

Recent advances in legged robotics have enabled their deployment in human-centered environments, expanding their roles from industrial inspection to service robotics and social interaction. At Chalmers University of Technology, the Robot Athletic Intelligence Lab (RAIL) has developed FIDO (Friendly and Interactive robot DOg), a quadruped robot based on the Unitree Go2 platform, designed to function as a campus tour guide. For robots like FIDO to interact effectively with humans in dynamic environments, rich multi-modal interaction, including nonverbal cues, is essential [1, 2, 3].

While speech is often treated as the primary auditory channel in human-robot interaction (HRI), nonverbal sound has increasingly been recognized as an important modality for communicating robot state, intention, and social presence [4]. However, motion-synchronous sound for legged robots remains comparatively underexplored, especially when the sound must be generated or triggered under embedded real-time constraints. In resource-constrained robotic systems, audio processing must share computational resources with navigation, perception, and control software. Consequently, many robotic performance systems either treat motion and sound as separate components or rely on pre-recorded music to which the robot synchronizes its movements [5].

This thesis investigates the inverse direction: an embedded system in which the robot’s physical movement drives musical feedback. This concept, referred to in this work as motion-driven sonification, introduces specific real-time engineering constraints. The system must react only to past and current sensor data, without look-ahead information, while handling natural variation in gait timing. It must also keep the system-induced delay between detected motion events and audio playback low enough to preserve the perceived connection between movement and sound.

### 1.1 Related Work

The intersection of robotics, music, and performance has been widely explored in the context of robotic dance. Prior work has demonstrated methods for mapping musical features to robot motion, enabling robots to execute choreographed movements synchronized to external music tracks [5, 6]. These systems are valuable for coordinated performance, but they typically follow a music-to-motion direction: the audio provides the temporal structure, and the robot adapts its movement to that structure.

In contrast, this thesis focuses on the opposite direction, where robot movement becomes the source of musical timing. Related work on nonverbal robot sound shows that robot-generated sound can influence user perception and can support more expressive interaction [4]. More directly, Cuan et al. [7] demonstrated a locomotion-driven framework that maps robot movement features to musical parameters, showing that movement-linked music can increase perceived intelligence and likability. This provides a key motivation for using motion-driven sound as an HRI modality.

A second challenge is computational. Large-scale generative and deep-learning-based audio systems can produce high-quality musical output, but their computational cost and inference latency may be unsuitable for tight interaction loops on embedded robotic platforms. Lightweight rule-based mappings and sample-based triggering are therefore attractive when predictable response time is more important than open-ended musical generation [8]. In this thesis, the online mode follows this design principle by using simple gait-event detection, tempo adaptation, and beat-aware sample scheduling.

Integrating such a system into a modern robotic software stack also presents timing challenges. ROS 2 was designed with real-time use cases in mind, but deterministic behavior still depends on middleware configuration, memory allocation, executor behavior, callback scheduling, and Quality of Service policies [9, 10, 11]. Standard Linux scheduling is optimized for fairness and throughput rather than bounded response time. At the operating-system level, `PREEMPT_RT` improves latency predictability by increasing kernel preemptibility, using priority-inheritance-aware locking, and threading interrupt handlers [12]. These mechanisms motivate the scheduler comparison performed in this thesis.

## 1.2 Purpose and Goal

**Purpose:** The wider purpose of this thesis is to explore how motion-synchronous audio can contribute to the expressive presence of an interactive robot. By enabling FIDO to generate sound in response to its own movement, the robot can provide an additional auditory channel through which nearby humans may perceive its motion state and rhythm. Such sonification may support more legible and engaging human-robot interaction, although a full psychological user study is outside the scope of this work.

**Goal:** The concrete technical goal of this project is to design, implement, and evaluate an embedded software module for the Unitree Go2 that extracts motion events from proprioceptive data and uses them to drive musical feedback. To accomplish this, the specific objectives include:

- **System Integration:** Extract gait timing and motion-event information from proprioceptive data using lightweight signal processing. The offline dance workflow uses multi-modal data, including inertial measurement unit (IMU), motor, and foot-force signals, while the online walking workflow focuses on foot-force-based gait-event detection using low-latency filtering and debounce logic.

- **Latency Optimization:** Keep the immediate, system-induced footstep-to-audio response below the perceptual synchrony target used in this work (target:  $< 150$  ms) [13]. This threshold is treated as an engineering design target rather than a universal perceptual boundary, since audiovisual synchrony judgments vary with stimulus type, timing direction, and event dynamics [14, 15]. Intentional beat-aligned musical deferrals are analyzed separately from system-induced latency.
- **Real-Time Scheduling:** Evaluate how OS-level real-time scheduling, using a `PREEMPT_RT` patched Linux kernel and `SCHED_FIFO` thread prioritization, affects callback timing, sample loss, and long-tail latency compared with standard Linux scheduling.

**Research Questions (RQs):** To systematically evaluate the achievement of these goals, this thesis addresses the following research questions:

- **RQ1:** How can proprioceptive sensor data from a quadruped robot be filtered and mapped to interactive musical feedback in both offline choreographed motion and online walking scenarios?
- **RQ2:** To what extent does OS-level real-time scheduling, using the `PREEMPT_RT` patch with `SCHED_FIFO` policy, reduce sample loss, callback jitter, and worst-case timing delays in an embedded audio-motion synchronization pipeline?

To ensure project feasibility, the evaluation focuses on technical metrics such as latency, sample loss, callback timing, and synchronization jitter. Extensive psychological user studies, professional musical quality assessment, and heavy AI-based music generation are kept out of scope.

## 1.3 Thesis Outline

The remainder of this thesis is structured as follows:

**Chapter 2 (Technical Background)** provides the theoretical foundation for the project, introducing the hardware architecture of the Unitree Go2, the principles of procedural audio and sonification, and the fundamentals of real-time operating systems and ROS 2 middleware.

**Chapter 3 (Method)** outlines the methodological approach adopted for this project. It introduces the overall system architecture, the core concepts of the sonification system, the experimental setup, and the primary metrics used to evaluate real-time performance.

**Chapter 4 (Design)** details the concrete software implementation and engineering evolution of the motion-driven sonification pipeline. It covers the data analysis, the transition from a Python prototype to a C++ online implementation, the workflows for both offline and online modes, and the OS-level integration strategies used to improve temporal predictability.

**Chapter 5 (Results)** presents the experimental data obtained from testing the system in both controlled scheduler experiments and real-world deployment on the

Jetson platform. It includes validation of the signal processing approach as well as statistical analyses of sample loss, execution jitter, callback timing, and end-to-end latency.

**Chapter 6 (Discussion)** interprets the experimental results, comparing the effectiveness of different real-time implementation strategies, discussing the robustness and limitations of the motion-feature extraction approach, and reflecting on the system's remaining constraints.

**Chapter 7 (Conclusion)** summarizes the main findings of the thesis, evaluates the fulfillment of the research goals, and suggests directions for future work.

# 2

## Technical Background

This chapter describes the theoretical concepts and underlying technologies required to understand the system presented in this thesis. It introduces the Unitree Go2 platform, quadrupedal locomotion and proprioceptive sensing, digital signal processing for motion-feature extraction, ROS 2 middleware, procedural audio and sonification, and the real-time operating-system concepts relevant to low-latency audio-motion synchronization.

### 2.1 Unitree Go2 Robot Dog

The Unitree Go2 is a quadruped robot platform developed by Unitree Robotics. The Go2 family includes several models, such as AIR, PRO, MAX, and EDU, with different hardware configurations and intended application scenarios [1].

The high-level motion-control interface of the robot can be categorized into two main types: basic motion control and special motion control. Basic motion-control services provide functions such as posture and body-height control, velocity control, trajectory tracking, and obstacle avoidance. In addition, the robot gait can be switched dynamically through interface calls according to terrain conditions and task requirements. Special motion-control services provide predefined skills embedded in the robot, such as sitting, jumping, and other demonstration motions [1].

Communication between functional modules of the robot is primarily implemented using the Data Distribution Service (DDS). DDS is a data-centric publish-subscribe middleware standard that supports decentralized communication and configurable Quality of Service (QoS) policies [16]. Unitree exposes robot state through DDS-compatible message definitions, which can also be accessed from ROS 2 through the Unitree ROS 2 interface [17]. In the Go2 low-state message, the available fields include IMU state, motor state, battery state, foot-force values, estimated foot-force values, wireless remote state, and a tick counter [18].

Users can subscribe to the DDS topic `rt/lowstate` to obtain system-state information, including motor status, battery data, IMU measurements, and remote-controller inputs. The IMU provides tri-axial acceleration and angular velocity measurements, as well as orientation information in the form of quaternions and Euler angles. The motor state provides joint-level information, while the foot-force channels provide direct information about foot-ground interaction.

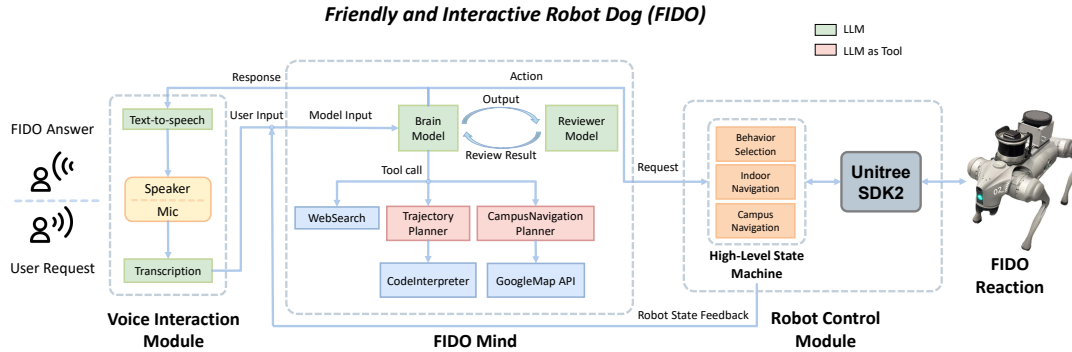
The `unitree_sdk2` software development kit provides APIs for low-level motor control, high-level motion control, radar point-cloud data, audio and video streaming,

SLAM, and odometry. The SDK provides interfaces in both C++ and Python, enabling secondary development of the Go2 platform based on vendor-provided examples and documentation [1].

The FIDO (Friendly and Interactive robot DOg) project builds upon the Unitree Go2 platform to create an autonomous campus tour-guide robot. To achieve autonomy in unstructured environments, the FIDO software ecosystem concurrently runs multiple computationally intensive ROS 2 nodes. These include a Simultaneous Localization and Mapping (SLAM) stack using 3D LiDAR data, a global and local navigation planner, and vision-based obstacle avoidance algorithms [3].

Figure 2.1 provides an overview of the FIDO system, including the voice interaction module, the FIDO Mind for reasoning and planning, and the robot control module. The robot control module interfaces with the Go2 through Unitree SDK2.

The simultaneous execution of these perception and navigation stacks places a heavy load on the robot’s onboard computing unit. This creates a contested computational environment where CPU resources and memory bandwidth are shared across navigation, perception, interaction, and secondary feedback modules. For an audio-motion synchronization subsystem to operate without interfering with locomotion and navigation, the implementation must keep the sensing and audio path lightweight and must explicitly consider scheduling behavior under load.



**Figure 2.1:** System architecture of the Friendly and Interactive Robot Dog (FIDO). Reproduced from Chen et al. [3].

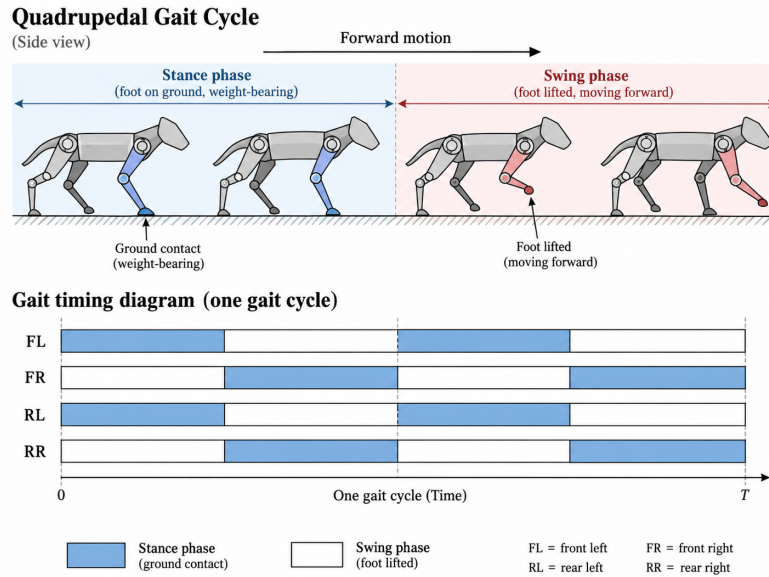
## 2.2 Quadrupedal Locomotion and Proprioceptive Data

In quadrupedal robots, locomotion is dynamic and requires continuous monitoring of the robot’s internal state. Proprioceptive sensing provides information about the robot body and joints, commonly through IMUs, joint encoders, joint torque or current estimates, and foot-contact or force-related signals [19, 20, 21].

### 2.2.1 Gait Cycle and Phases

A quadrupedal gait describes the temporal coordination of foot movements. For each leg, the locomotion cycle can be divided into two fundamental phases, as illustrated in Figure 2.2:

- **Stance phase:** the foot is in contact with the ground, supporting the robot's weight and generating propulsion.
- **Swing phase:** the foot is lifted and moves forward in preparation for the next ground contact.



**Figure 2.2:** Illustration of a standard quadrupedal gait cycle, highlighting the transition between the stance phase (ground contact) and the swing phase. AI-generated using GPT-6 (OpenAI).

The transition between these phases, particularly touchdown, lift-off, and unloading events, plays an important role in locomotion analysis and control. Contact timing provides discrete temporal markers that can be used for gait-phase estimation, state estimation, and event-driven feedback [22, 19, 21]. In this thesis, such gait-related events are used as timing cues for motion-driven audio feedback rather than for closed-loop locomotion control.

### 2.2.2 Proprioceptive Sensors and Data Sources

Modern quadrupedal platforms integrate multiple proprioceptive sensors to describe their internal state:

- **Inertial Measurement Unit (IMU):** provides tri-axial acceleration and angular velocity, which reflect body-level motion such as torso sway, impact, roll, pitch, and yaw.
- **Joint encoders and motor-state estimates:** measure or estimate joint

positions, velocities, and related motor quantities at high frequency, enabling tracking of limb kinematics.

- **Force-related signals:** describe foot-ground interaction. Ground contact can be detected from dedicated foot-force sensors or inferred indirectly from actuator signals, estimated ground reaction forces, or kinematic and dynamic consistency checks [22, 21].

In full-body state-estimation systems, these signals are often fused to estimate base pose, velocity, contact states, and foothold locations [19, 20]. The present thesis uses a narrower subset of this information. The offline dance workflow extracts motion features from IMU, motor, and foot-force signals, while the online walking workflow prioritizes low-latency foot-force-based event detection.

### 2.3 Signal Processing

In practical robotic scenarios, raw sensor signals are often contaminated by high-frequency noise originating from mechanical vibration, actuator dynamics, impact transients, and uneven terrain. Digital signal processing is therefore required to extract underlying kinematic patterns reliably while preserving timing information relevant to real-time interaction.

Digital filters can generally be categorized as Infinite Impulse Response (IIR) filters and Finite Impulse Response (FIR) filters. IIR filters include feedback terms and therefore have an impulse response of theoretically infinite duration. FIR filters are purely feedforward structures with a finite impulse response [23, 24]. In the  $z$ -domain, this distinction appears in the transfer function: IIR filters include both numerator and denominator polynomials, while FIR filters contain only numerator terms.

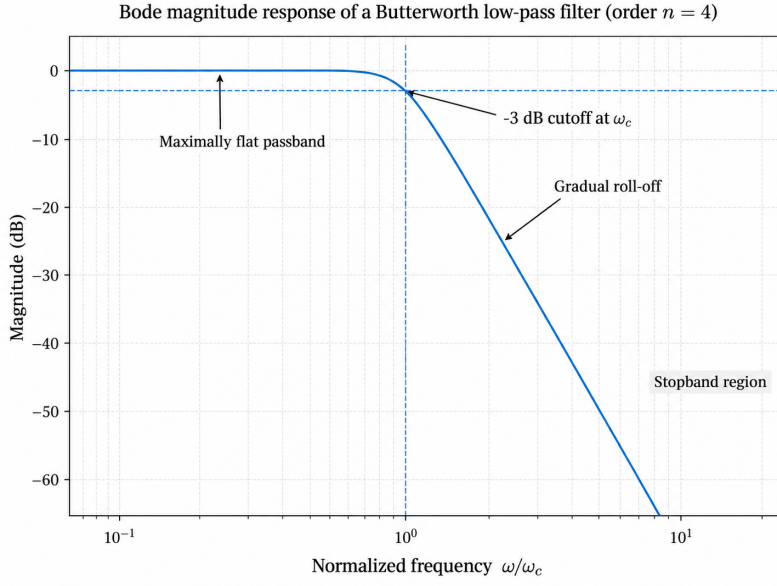
IIR filters are often useful in resource-constrained and real-time systems because they can achieve a desired frequency response with a lower filter order than an FIR filter with comparable roll-off. This reduces memory use and per-sample computation, although care must be taken with phase delay, initialization transients, and numerical stability [23, 24].

#### 2.3.1 Butterworth Low-Pass Filter Theory

The Butterworth filter is an IIR filter family designed to have a maximally flat magnitude response in the passband [25]. This property preserves the amplitude of low-frequency motion components without passband ripple, making it suitable for smoothing noisy motion signals when sharp passband ripple would distort the extracted movement features (Figure 2.3).

The continuous-time transfer function of a typical fourth-order Butterworth low-pass filter can be expressed as:

$$H(s) = \frac{\omega_c^4}{(s^2 + a_1\omega_c s + \omega_c^2)(s^2 + a_2\omega_c s + \omega_c^2)} \quad (2.1)$$



**Figure 2.3:** Typical frequency response of a Butterworth low-pass filter, demonstrating the maximally flat passband and gradual roll-off. AI-generated using GPT-6 (OpenAI). The filter theory is described in [25].

where  $H(s)$  denotes the continuous-time transfer function,  $s$  is the complex Laplace variable,  $\omega_c = 2\pi f_c$  is the cutoff angular frequency,  $f_c$  is the cutoff frequency in Hz, and  $a_1$  and  $a_2$  are the second-order section coefficients determined by the fourth-order Butterworth polynomial.

For digital signal processing, the continuous-time filter must be converted into a discrete-time representation, commonly using the bilinear transform [24]. Direct implementation of a high-order IIR filter using a single transfer-function polynomial can be numerically sensitive. Therefore, high-order IIR filters are commonly implemented as cascaded Second-Order Sections (SOS):

$$H(z) = H_1(z) \cdot H_2(z) \quad (2.2)$$

where each section is given by:

$$H_i(z) = \frac{b_{i0} + b_{i1}z^{-1} + b_{i2}z^{-2}}{1 + a_{i1}z^{-1} + a_{i2}z^{-2}}, \quad i = 1, 2 \quad (2.3)$$

where  $H(z)$  denotes the overall discrete-time transfer function,  $z^{-1}$  represents a one-sample delay, and  $H_i(z)$  is the transfer function of the  $i$ -th second-order section. The coefficients  $b_{i0}$ ,  $b_{i1}$ , and  $b_{i2}$  are the numerator coefficients, while  $a_{i1}$  and  $a_{i2}$  are the denominator coefficients of the  $i$ -th section. This SOS representation improves numerical robustness and reduces sensitivity to coefficient quantization errors [23, 26].

### 2.3.2 Filter Group Delay and Transient Response

Implementing filters in causal recursive form for real-time processing introduces dynamic effects. As a linear time-invariant (LTI) system, an IIR filter introduces a

frequency-dependent *group delay*, defined as:

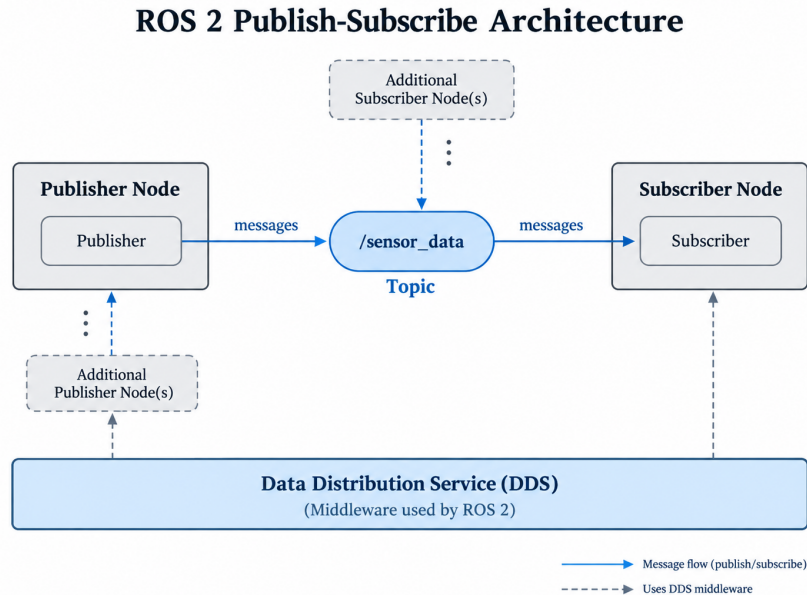
$$\tau_g(\omega) = -\frac{d\phi(\omega)}{d\omega} \quad (2.4)$$

where  $\tau_g$  is the group delay,  $\omega$  is the angular frequency, and  $\phi(\omega)$  denotes the phase response of the filter [23]. A lower cutoff frequency improves noise attenuation but increases group delay, which can be problematic for online interaction because it adds latency between the physical motion and the detected event.

Recursive filters also exhibit a *transient response* during initialization or rapid input changes. The transient component typically decays according to the pole locations of the filter. In real-time engineering, an initial warm-up period is often used to allow the filter state to settle before its output is used for event detection.

## 2.4 Robot Operating System 2 (ROS 2)

The Robot Operating System 2 (ROS 2) is a widely used middleware framework for modern robotics. It provides a modular distributed architecture in which components communicate through topics, services, and actions. ROS 2 was redesigned from ROS 1 to better support production robotics requirements, including decentralized communication, improved security support, and integration with DDS middleware [9]. The basic publish—subscribe communication model in ROS 2 is illustrated in Figure 2.4.



**Figure 2.4:** Basic publish-subscribe communication model in ROS 2 utilizing Data Distribution Service (DDS). Drawn by the authors based on the architecture described in [9].

### 2.4.1 DDS-based Communication Model

Unlike ROS 1, ROS 2 is built upon DDS, which provides decentralized data-centric publish-subscribe communication, discovery, and configurable QoS policies [9, 16]. QoS policies such as reliability, deadline, history, durability, and lifespan allow developers to tune communication behavior for different robotic workloads. These settings are useful for balancing freshness, reliability, and resource use, especially in high-rate sensing pipelines.

However, DDS support does not by itself guarantee deterministic timing for every ROS 2 application. End-to-end behavior also depends on middleware implementation, QoS configuration, network conditions, callback scheduling, memory behavior, and operating-system scheduling. Therefore, real-time robotic applications must evaluate both middleware configuration and application-level execution behavior under realistic load.

### 2.4.2 Execution Model and Executors

ROS 2 uses an executor-based execution model to invoke callbacks associated with subscriptions, timers, services, action servers, and other events. An executor uses one or more operating-system threads to wait for work and execute callbacks [27]. Executors can be single-threaded, where callbacks are processed sequentially, or multi-threaded, where several callbacks can run concurrently.

To control concurrency, developers use *callback groups*:

- **Mutually Exclusive Callback Group:** callbacks assigned to this group are not executed concurrently.
- **Reentrant Callback Group:** callbacks within the group may be executed simultaneously by different executor threads.

The scheduling semantics of ROS 2 executors are important for real-time analysis. When callbacks take longer than the period at which messages arrive, messages may queue in lower layers of the middleware stack, and the executor does not necessarily process them in strict FIFO order [28, 27]. For high-rate sensing and audio-triggering pipelines, this means that callback timing and queuing behavior must be measured rather than assumed.

### 2.4.3 Real-Time Challenges in ROS 2

Although ROS 2 improves upon ROS 1, it is not inherently a hard real-time framework [29]. Sources of nondeterminism include callback scheduling, thread contention, dynamic memory allocation, page faults, blocking synchronization, middleware queues, and operating-system preemption [10, 11, 28]. These factors can introduce jitter and long-tail delays even when the average computational cost of a callback is small.

For the audio-motion synchronization problem in this thesis, the relevant requirement is not hard real-time locomotion control. Instead, the goal is strict soft real-time behavior: the system should keep motion-triggered audio feedback below per-

ceptual timing limits for immediate events, while avoiding rare scheduler delays that cause missed sensor updates or delayed audio-buffer insertion.

## 2.5 Procedural Audio and Sonification

Sound generation in interactive systems can be broadly categorized into sample-based playback and procedural synthesis. Sample-based playback triggers pre-recorded audio clips, while procedural synthesis generates or modifies audio algorithmically in response to changing input parameters [8].

### 2.5.1 Procedural Audio Synthesis and Sonification

Procedural audio refers to sound generated or controlled in real time by algorithmic processes rather than by fixed playback alone. This approach is common in interactive applications because synthesis parameters can be modified continuously according to user actions or sensor data [8].

Sonification is the translation of data into non-speech sound for communication, monitoring, exploration, or interaction [30, 31]. In human-robot interaction, non-verbal sound has been identified as an important modality for robot expression and communication, although terminology and evaluation methods vary across the field [4]. Mapping physical movement to sound can therefore create an acoustic layer through which human observers perceive the robot’s activity and rhythm [32, 7].

Human psychophysical studies show that perceived audio-visual synchrony depends on stimulus type, content, and whether sound leads or lags visual motion. Van Eijk et al. report that audio-visual pairs are often judged synchronous over a window from approximately  $-50$  ms audio lead to  $+150$  ms video lead [13]. Other studies similarly show that dynamic audiovisual events and action-related sounds can produce different synchrony judgments depending on event structure and context [14, 15]. In this thesis, the 150 ms value is therefore used as an engineering design target for immediate footstep-to-audio feedback, not as a universal perceptual boundary.

### 2.5.2 Fundamentals of Musical Parameters

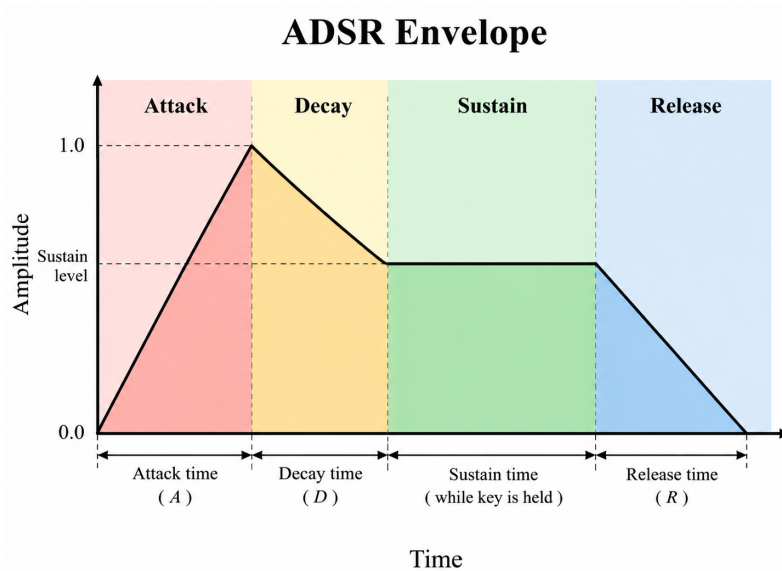
To map mechanical motion to sound, several fundamental musical dimensions can be manipulated:

- **Tempo (BPM):** Beats Per Minute defines the speed or pace of a rhythmic sequence.
- **Pitch (Frequency):** the perceived highness or lowness of a sound.
- **Velocity (Intensity):** in musical terminology, velocity refers to how strongly a note is struck, often mapped to loudness or amplitude.

For real-time robotic applications, these parameters must be selected so that they can be updated with predictable computational cost. The online system in this thesis therefore uses lightweight rule-based mappings rather than heavy generative audio models.

### 2.5.3 Amplitude Envelopes (ADSR)

In procedural audio, raw continuous waveforms are shaped over time to sound natural, particularly when simulating discrete transient events. This is commonly done using an amplitude envelope, such as the ADSR (Attack, Decay, Sustain, Release) model shown in Figure 2.5 [8].



**Figure 2.5:** The ADSR (Attack, Decay, Sustain, Release) amplitude envelope used to shape the volume of procedural sound over time. AI-generated using GPT-6 (OpenAI). The ADSR model is discussed by Pillay [33].

The *Attack* phase defines the time required for the sound to reach its maximum amplitude. The *Decay* phase determines how quickly the sound drops to the *Sustain* level, which is the steady amplitude maintained while the note or trigger is held. The *Release* phase specifies how long the sound takes to fade after the trigger ends. Without suitable envelope shaping, abrupt starts and stops in a waveform can create discontinuities that are perceived as clicks or pops.

## 2.6 Real-Time Operating Systems (RTOS) Concepts

A general-purpose operating system such as standard Ubuntu Linux is designed to balance throughput, fairness, responsiveness, and resource sharing among many processes. Since Linux 2.6.23, the default Linux scheduler has been the Completely Fair Scheduler (CFS), exposed to normal user processes through the `SCHED_OTHER` scheduling policy [34]. This policy is effective for general workloads, but it does not guarantee that a time-critical callback will run immediately when it becomes ready.

### 2.6.1 Hard vs. Soft Real-Time Systems

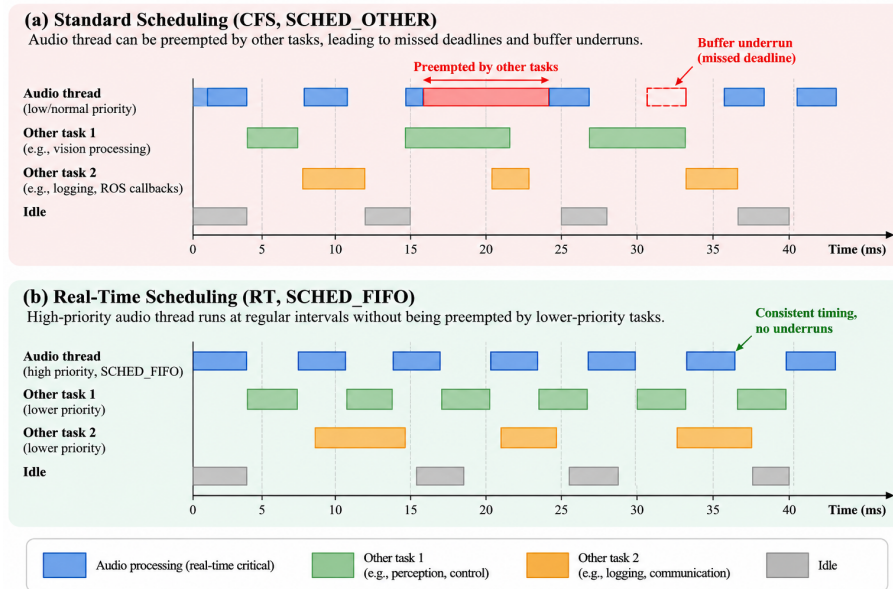
Real-time systems are commonly described according to their tolerance to missed deadlines:

- **Soft real-time:** missing a deadline degrades quality of service but does not constitute total system failure, such as a dropped audio frame or delayed user-facing feedback.
- **Hard real-time:** missing a deadline is treated as system failure, such as in a safety-critical control loop where late control updates can threaten stability.

This thesis does not attempt to validate safety-critical locomotion control. Instead, the audio subsystem is treated as a strict soft real-time workload. A missed or delayed audio event may reduce interaction quality, but it is not part of the robot’s balance or safety controller.

### 2.6.2 Audio Buffering and Kernel-Level Scheduling

Digital audio systems process sound in fixed-size buffers. The audio hardware consumes these buffers at a fixed sample rate, and the audio callback must fill each buffer before it is needed. If the callback is delayed for too long, a buffer underrun may occur, causing audible glitches or dropouts. Figure 2.6 provides a conceptual comparison of audio-thread scheduling under CFS/`SCHED_OTHER` and fixed-priority `SCHED_FIFO`.



**Figure 2.6:** Conceptual comparison of CFS/`SCHED_OTHER` and fixed-priority `SCHED_FIFO` scheduling. AI-generated using GPT-6 (OpenAI); not a measured execution trace. Scheduling policies and real-time latency are described in [34, 35]. The illustrated reduction in buffer underruns is not a guarantee.

Linux provides real-time scheduling policies such as `SCHED_FIFO` and `SCHED_RR`. Under `SCHED_FIFO`, a runnable real-time thread with higher static priority can pre-

empt normal `SCHED_OTHER` threads and will continue running until it blocks, yields, or is preempted by an even higher-priority thread [34]. This behavior is useful for time-critical audio and sensing callbacks, but it must be used carefully because a high-priority thread that fails to block or yield can starve lower-priority work.

`PREEMPT_RT` improves Linux real-time behavior by increasing kernel preemptibility, moving interrupt handling into schedulable thread context, and replacing certain non-preemptible locking paths with priority-inheritance-aware mechanisms [12, 36]. These changes reduce sections of kernel execution in which a higher-priority task must wait for non-preemptible work to finish.

Priority inversion is another important concern in multi-threaded real-time systems. It occurs when a high-priority thread is blocked on a lock held by a lower-priority thread, while intermediate-priority work prevents the lower-priority lock holder from running. Priority-inheritance locking mitigates this by temporarily boosting the lock holder so it can release the lock more quickly [12]. This concept is directly relevant to the audio-motion pipeline in this thesis, where a sensing callback and an audio callback share a small amount of state.

In summary, real-time scheduling does not make the audio or signal-processing algorithm faster in the average case. Its main value is to improve timing predictability by reducing long-tail scheduling delays, missed high-rate callbacks, and late audio-buffer insertion under competing system load.



# 3

## Method

This chapter outlines the methodological approach adopted in this thesis to design, implement, and systematically evaluate a motion-driven audio generation system in a ROS 2 and Ubuntu-based environment. The development and deployment are conducted in an Ubuntu 20.04 environment running ROS 2 Foxy. The methodology consists of the development of a sensor processing pipeline, including signal filtering and motion event detection, as well as the integration of real-time audio triggering mechanisms.

In addition, a layered analysis framework is proposed to investigate timing behavior across the operating system, ROS 2 middleware, and application levels. Real-time performance is assessed using end-to-end latency and jitter as primary metrics, while their contributing factors are analyzed through controlled experiments and specific timestamp instrumentation within the ROS 2 node.

### 3.1 System Overview

The overall system architecture is illustrated in Figure 3.1. The proposed system converts robot motion into audio signals in real time through a multi-stage processing pipeline.



**Figure 3.1:** Sonification Pipeline. AI-generated using GPT-6 (OpenAI).

The system consists of three main modules: signal processing, motion feature extraction, and music synthesis. Sensor data, including IMU measurements, joint states and foot force data, are first processed to reduce noise and extract meaningful motion features. These features are then mapped to musical elements, which are

synthesized and played through an audio output system. All functional testing is conducted in an indoor laboratory environment, utilizing an external loudspeaker for audio playback.

This pipeline enables dynamic and responsive sonification of the robot’s motion.

## 3.2 Sonification System

The sonification system transforms processed motion data into audio signals. Among the entire pipeline, two key components require careful design: signal filtering and motion-to-sound mapping.

### 3.2.1 Signal Filtering

To suppress high-frequency noise in the motor signals, a low-pass filtering strategy is applied. Based on prior analysis, a Butterworth filter is adopted due to its smooth passband response and suitability for real-time applications.

The selection of the cutoff frequency is guided by both frequency-domain analysis and empirical validation. First, spectral analysis is used to identify the dominant frequency range of motion-related signals, which provides an initial estimate of the cutoff frequency. This value is then refined based on the observed filtering performance in the time domain to ensure an appropriate balance between noise suppression and signal fidelity.

The cutoff frequency reflects a trade-off between noise suppression and latency: lower values improve smoothing but increase delay, while higher values reduce delay at the cost of more noise. Therefore, the cutoff frequency is chosen to balance filtering effectiveness and real-time responsiveness.

### 3.2.2 Motion-to-Sound Mapping and Music Generation

An action is identified from the filtered signal when its amplitude exceeds a pre-defined threshold, followed by a debounce condition to ensure robustness against transient fluctuations.

The extracted motion features are mapped to musical elements using a rule-based strategy. Different motion characteristics are associated with distinct sound layers, such as rhythm, percussive elements, or background effects. To ensure the generated audio sounds natural and avoids audible artifacts like zero-crossing clicks, specific ADSR (Attack, Decay, Sustain, Release) amplitude envelopes are applied depending on the physical nature of the movement:

- **Percussive Envelopes (Foot Strikes):** When the robot’s foot strikes the ground (e.g., the start of the stance phase), the impact is mapped to a percussive envelope. This is characterized by a near-instantaneous *Attack* (rapid rise in volume) followed by a swift *Decay/Release*, mimicking the sharp, discrete sound of a physical impact.

- **Continuous Envelopes (Body Sway/Sliding):** Conversely, continuous states like the robot’s torso sway or forward velocity are mapped to sounds with longer *Attack* and *Sustain* phases. These sounds gradually fade out (*Release*) only when the continuous motion comes to a halt.

Following this mapping, a music layer selection module determines which sound components are activated based on the current motion state. The selected layers are then passed to a music generator, which synthesizes audio signals using a predefined sample bank. This design enables dynamic and responsive audio generation that vividly reflects the robot’s physical motion in real time.

### 3.3 Real-Time System Design

To ensure low-latency and synchronized audio generation, real-time performance is strictly addressed at the operating system level. Standard Linux systems prioritize overall throughput and fairness, which can introduce non-deterministic preemption and unacceptable delays for audio-motion synchronization. At the system level, real-time support is enabled by applying the `PREEMPT_RT` patch to the Linux kernel. This transforms the standard Linux system into a real-time operating environment with deterministic scheduling behavior. By configuring real-time priorities (e.g., assigning higher POSIX priorities to the audio synthesis thread), the system ensures that critical tasks are executed within bounded time constraints, effectively minimizing OS-level jitter and preventing audio buffer underruns.

### 3.4 Experimental Setup and Hardware Constraints

A robust evaluation of the real-time system requires comparing the performance of the standard Linux kernel against the `PREEMPT_RT` patched kernel. However, modifying the kernel of the Unitree Go2’s onboard computer presents significant operational risks. The robot serves as a shared hardware platform for at least three concurrent research project groups. Flashing a custom kernel introduces the risk of bricking the onboard system, which would require returning the robot to the manufacturer for repairs, severely disrupting all ongoing projects. To mitigate this risk while still rigorously evaluating the performance metrics of the proposed ROS 2 nodes and the impact of the real-time patch, a surrogate evaluation platform was introduced. A Raspberry Pi running Linux and ROS 2 was set up as an isolated testing environment. The comparative experiments—measuring system latency and jitter with and without the `PREEMPT_RT` patch—were executed on this Raspberry Pi platform. This approach ensures the safety of the robot while providing valid, hardware-isolated benchmarking data for the software architecture.

### 3.5 Metrics for Real-Time Performance

To evaluate the real-time performance of the proposed system, timing behaviors are analyzed directly at the application and middleware interaction level. The overall

system performance is evaluated using two primary end-to-end metrics:

- **End-to-End Latency:** Defined as the time elapsed from the moment motion data is received to the moment the corresponding audio signal is ready for playback. Latency is empirically measured using a precise timestamping mechanism injected directly into the ROS 2 node. Specifically, the initial timestamp( $T_{start}$ ) is recorded at the exact moment the system enters the sensor data callback function. After the signal processing, motion extraction, and music synthesis stages are completed, a final timestamp( $T_{end}$ ) is recorded precisely at the moment the filled audio buffer is dispatched to the external speaker. The latency is calculated as  $\Delta T = T_{end} - T_{start}$ .
- **System Jitter:** Defined as the temporal instability or variation of the end-to-end latency over continuous operation. In this thesis, jitter is quantified by calculating the statistical variance (and standard deviation) of a large set of measured latencies across repeated execution cycles. High variance indicates significant timing unpredictability, which degrades the rhythmic consistency of the sonification.

To quantify the benefits of the real-time optimizations, a strict comparative experiment is designed. The system is run under two distinct configurations on the Raspberry Pi evaluation platform:

- **Baseline Configuration:** The ROS 2 audio node running on a standard, unpatched Ubuntu 22.04 Linux kernel.
- **Real-Time Configuration:** The same ROS 2 audio node running on the Ubuntu Linux kernel modified with the `PREEMPT_RT` patch.

By comparing the latency distribution and jitter variance between these two configurations, the necessity and effectiveness of kernel-level real-time patching for robotic audio-motion synchronization can be conclusively evaluated.

# 4

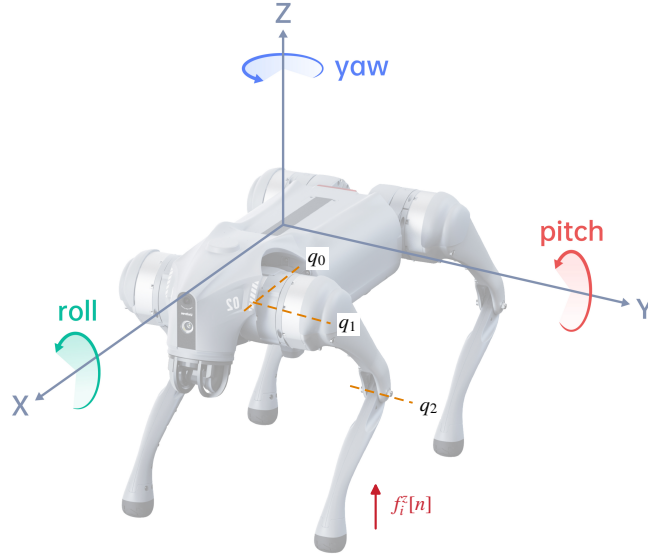
## Design

This chapter details the system design and software implementation of the motion-driven sonification pipeline. It first introduces the robot sensor data used for motion extraction, then explains the offline dance-sonification workflow, the online walking-sonification algorithm, and the real-time integration strategy used on the deployed FIDO robot.

### 4.1 Data Analysis and Motion Extraction

#### 4.1.1 Robot Original Data

Foot force, IMU, and motor state information are acquired through the Unitree DDS low-state data stream. The three-dimensional coordinate frames of the robot are illustrated in Figure 4.1.



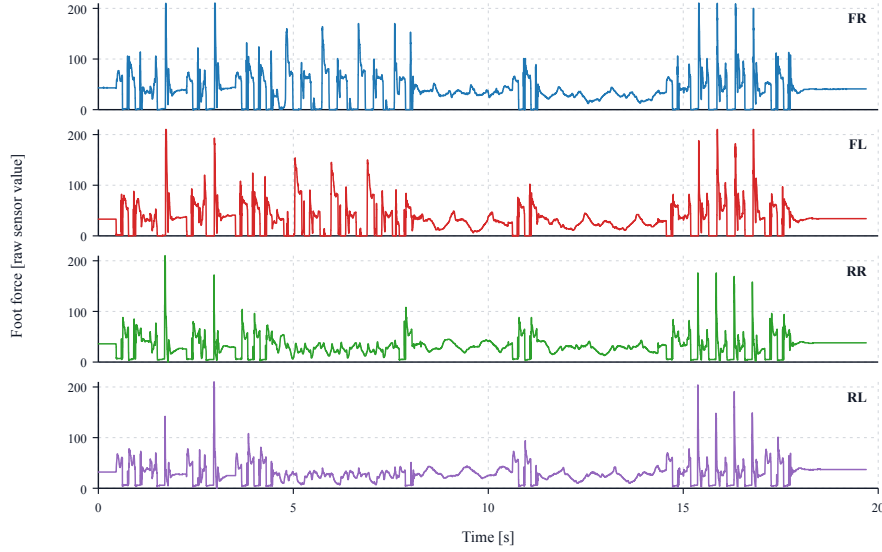
**Figure 4.1:** Coordinate axes, force directions, and sensor annotations for the Unitree Go2. Base robot image from Unitree Robotics [37]; coordinate axes, force arrows, and sensor annotations added by the authors.

The foot force data represent the ground reaction force measured at each leg. The IMU provides multiple types of state information, including quaternion data, Euler

angles, angular velocity, and linear acceleration along the roll, pitch, and yaw axes.

A total of 12 motors are available, corresponding to the four legs of the robot. Each leg is actuated by three joints: hip, thigh, and calf. For each motor, the following feedback signals are provided: joint angular position, angular velocity, angular acceleration, and torque.

The foot force signals are utilized for gait extraction, and the raw data quality is sufficient for reliable feature extraction, as shown in Figure 4.2.

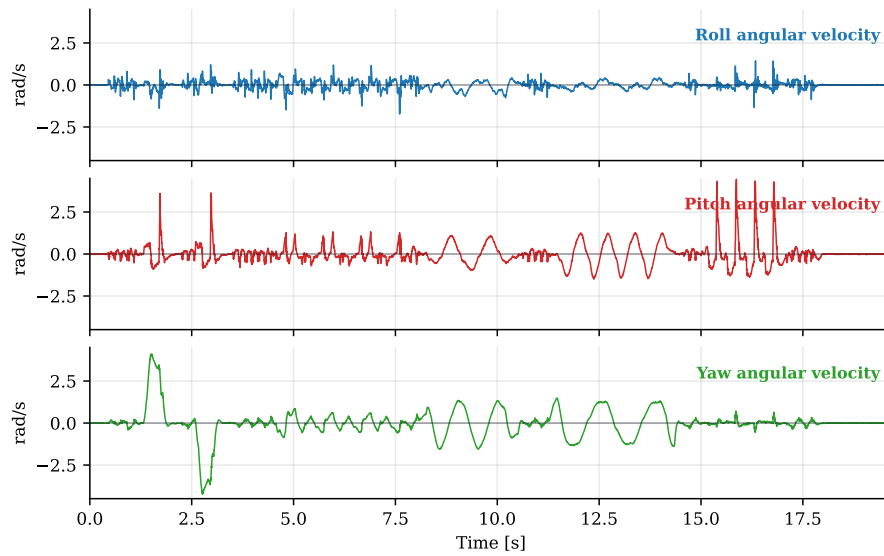


**Figure 4.2:** Original foot force sensor data.

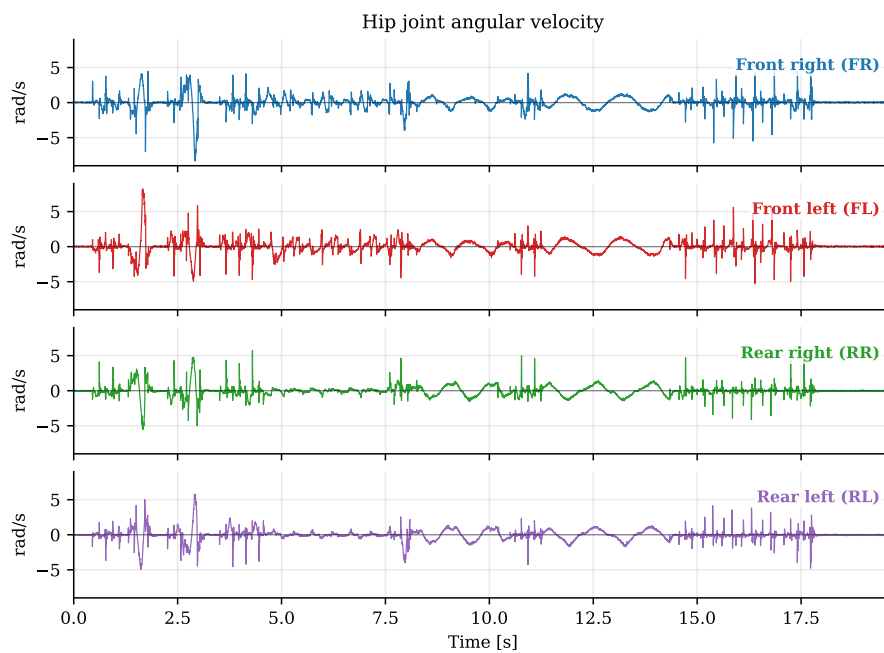
Angular velocity serves as a key metric for representing the robot’s motion. The IMU-derived angular velocity is relatively smooth, as the mechanical structure of the legs acts as a passive damping system. Consequently, only limited noise is present in the raw IMU signals, making them suitable for extracting body motion features. The original IMU signals are plotted in Figure 4.3.

In contrast, the motor angular velocity signals exhibit significant noise. Notably, the joint velocity is not computed as a numerical derivative of the angular position. Instead, it is an internal estimate provided by the motor controller, typically derived from observer-based estimation or velocity-loop signals within a field-oriented control (FOC) framework. To improve readability, the 12 motor angular velocity signals are divided into three groups according to joint type: hip, thigh, and calf. The original motor angular velocity signals for these three joint groups are shown in Figures 4.4, 4.5, and 4.6, respectively.

As a result, the angular velocity is highly sensitive to small, high-frequency micro-motions caused by ground contact, load redistribution, and structural compliance. Large instantaneous fluctuations in velocity may occur even when the corresponding joint angle variations are minimal. Therefore, appropriate signal processing techniques are required to filter the motor angular velocity data and reliably extract leg motion features.



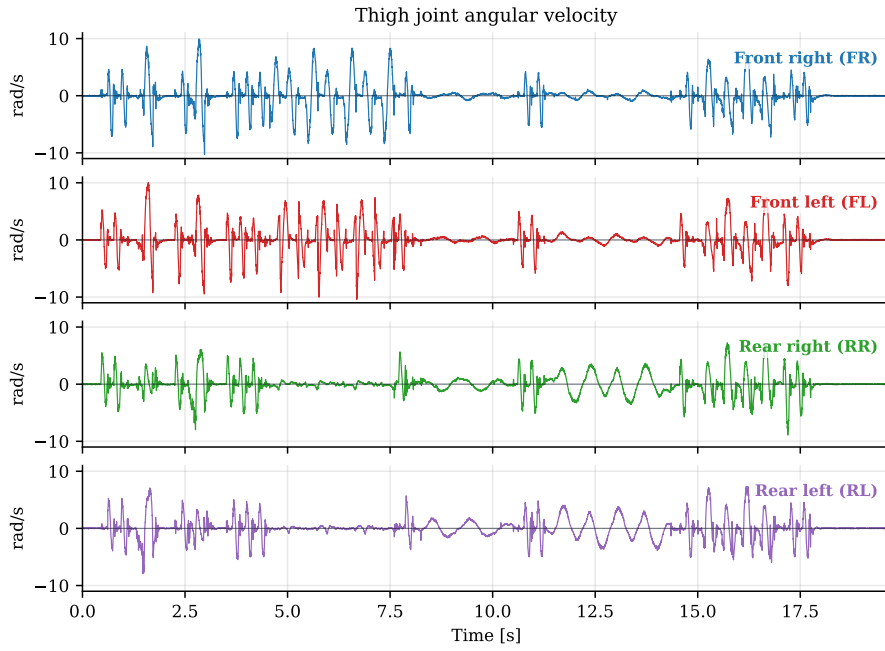
**Figure 4.3:** Original IMU data representing body motion.



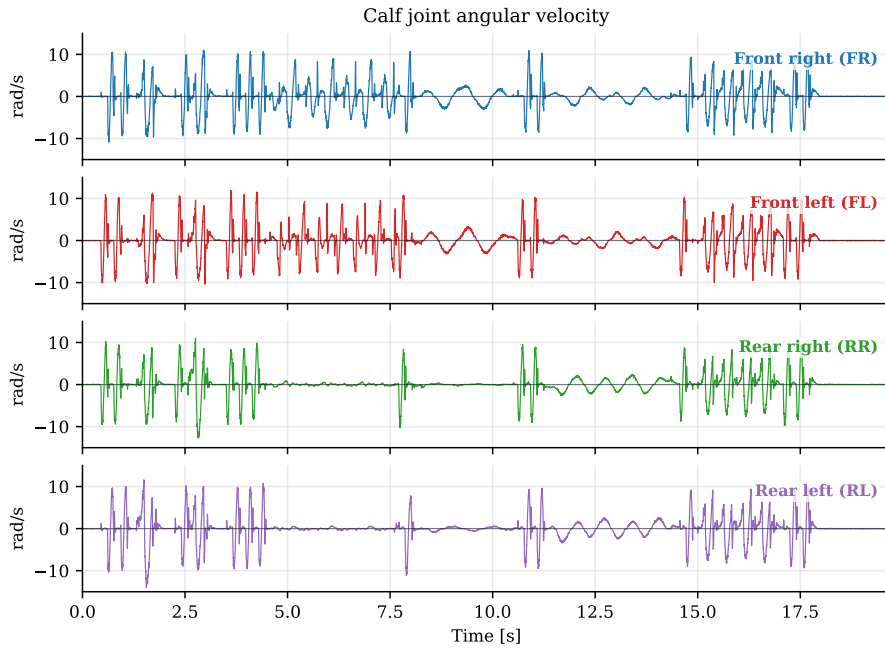
**Figure 4.4:** Original hip joint motor angular velocity data exhibiting significant noise.

#### 4.1.2 Butterworth Filter Cutoff Frequency

The appropriate cutoff frequency must be carefully determined to ensure that motion features within the relevant frequency range can be effectively extracted. To achieve this, a frequency-domain analysis of the motor signals was first conducted. As illustrated in Figure 4.7, the spectral characteristics of the motor data allow for the identification of a suitable cutoff frequency range for subsequent signal processing.



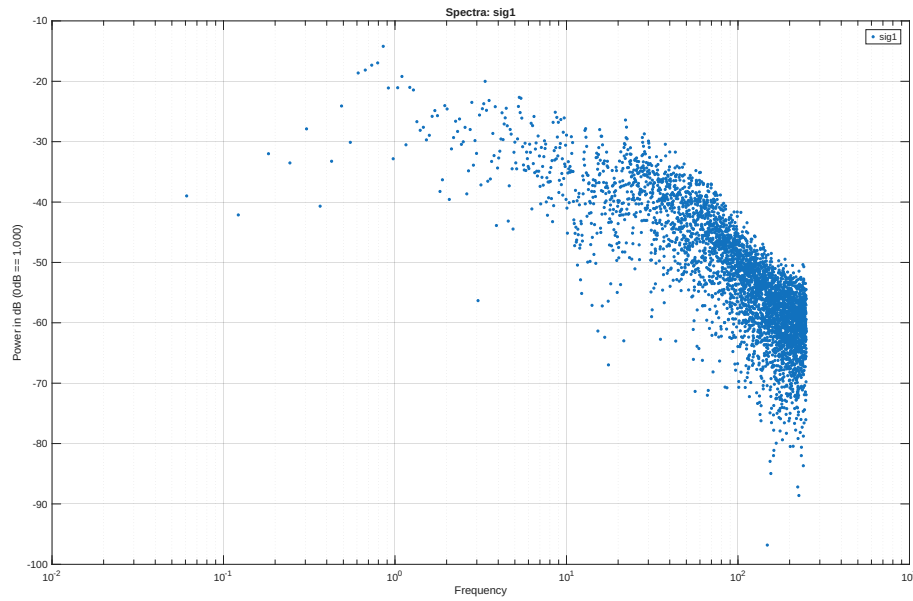
**Figure 4.5:** Original thigh joint motor angular velocity data exhibiting significant noise.



**Figure 4.6:** Original calf joint motor angular velocity data exhibiting significant noise.

### 4.1.3 Butterworth Filter Group Delay

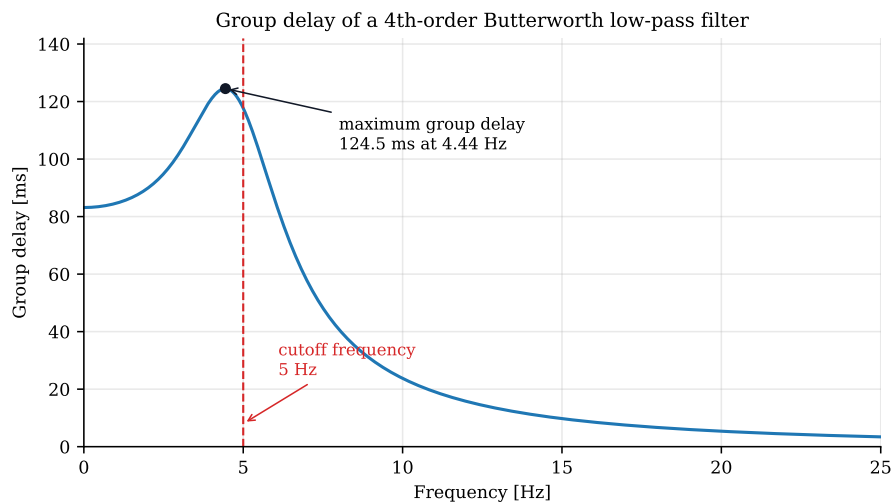
In this work, a fourth-order Butterworth low-pass filter is employed to suppress high-frequency noise in the joint velocity signals. The filter group delay depends on both the filter order and the cutoff frequency. Since the filter order is fixed,



**Figure 4.7:** Spectral analysis of the FR hip motor angular velocity.

the cutoff frequency becomes the dominant factor affecting the delay. Lower cutoff frequencies provide stronger noise attenuation but introduce larger temporal delays.

For the sampling frequency of 503.018 Hz used in this work, the measured maximum group delay is approximately 124 ms for a cutoff frequency of 5 Hz, as illustrated in Figure 4.8.



**Figure 4.8:** Butterworth filter group delay with cutoff frequency 5 Hz.

#### 4.1.4 Butterworth Filter Transient Response Analysis

The filter was implemented as a causal recursive infinite impulse response (IIR) filter for real-time processing. To ensure numerical stability, the filter was realized in second-order section (SOS) form. The transient response of the Butterworth filter

is governed by the real part of its dominant pole. The filter cutoff frequency was set to  $f_c = 5$  Hz, and the sampling frequency was  $F_s = 503$  Hz. For a fourth-order Butterworth filter, the slowest pole satisfies:

$$\sigma = 0.3827 \omega_c \quad (4.1)$$

where

$$\omega_c = 2\pi f_c \quad (4.2)$$

is the cutoff angular frequency. The corresponding time constant is given by:

$$\tau = \frac{1}{\sigma} \quad (4.3)$$

Substituting  $f_c = 5$  Hz yields:

$$\tau \approx 83.2 \text{ ms.} \quad (4.4)$$

The transient response typically decays within approximately  $3\tau$  to  $5\tau$ , corresponding to a duration of approximately 250–416 ms. Given the sampling frequency of 503 Hz, this corresponds to approximately 125–208 samples. Therefore, a warm-up period of approximately 200 samples was applied before using the filtered signal for subsequent processing, ensuring that the transient effects are sufficiently suppressed.

## 4.2 Software Evolution: From Python to C++

The development of the sonification software underwent a two-phase evolution to meet the strict latency requirements of real-time robotics.

### 4.2.1 Prototyping Phase (Python)

In the initial development phase, the motion-to-sound mapping system was prototyped using Python to facilitate rapid development and offline data verification. The Python implementation utilized the `sounddevice` library for audio playback and the `audiotsm` package to manipulate music tempo dynamically. While this architecture successfully validated the mapping algorithms offline, transitioning it to a real-time online node exposed severe performance bottlenecks. The Python Global Interpreter Lock (GIL), garbage collection, and high-level audio callback overhead introduced unpredictable latency spikes and audio buffer underruns, making the Python version unsuitable for synchronized live locomotion.

### 4.2.2 Production Phase (C++)

To address the performance constraints, the online node was rewritten in C++. The production architecture leverages low-level libraries and multi-threading capabilities to ensure deterministic audio generation.

The C++ audio engine is built upon two core libraries:

- **SDL2:** Used as the hardware abstraction layer to interact directly with the audio device. SDL provides an efficient callback-driven audio thread that repeatedly requests fixed-size output buffers.

- **Signalsmith Stretch:** A C++ audio time-scale modification library used to stretch or compress the playback speed of the background music without altering pitch. This replaces the Python `audiotism` package in the online implementation.

## 4.3 Offline Mode

### 4.3.1 Rationale for the Offline Baseline

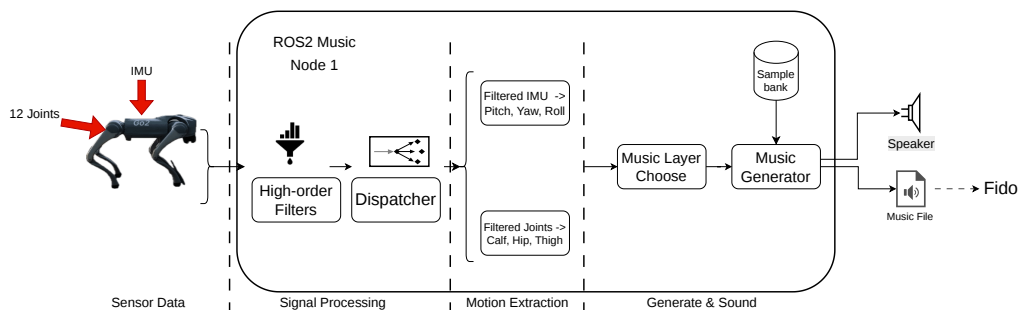
The Offline Mode was developed for predefined robot dance routines, where the robot motion sequence is known before playback. This makes it possible to prioritize musical quality and precise alignment over real-time computational constraints. The purpose of this mode is not to replace online walking sonification, but to provide a high-fidelity baseline for complex choreographed motions.

The offline pipeline serves two methodological purposes. First, it isolates the validation of the motion-to-sound mapping logic from operating-system scheduling effects. Since the data are processed after recording, the pipeline can use computationally heavier fourth-order Butterworth filters, multiple sensor modalities, and manual DAW-based music engineering. Second, it supports complex dance gestures whose musical accompaniment benefits from harmonic and structural editing that would be too expensive to synthesize on the embedded robot computer in real time.

The main drawback of this approach is that the music cannot react to unexpected changes in the robot's motion. Instead, the robot executes a predefined dance, the motion events are extracted from a recorded run, and the final audio is rendered in advance. During robot operation, playback is triggered at the start of the corresponding dance routine.

### 4.3.2 Offline Mode Workflow

The complete Offline Mode workflow is shown in Figure 4.9. It consists of three steps: data acquisition and filtering, music engineering and stem mapping, and final rendering and embedded deployment.



**Figure 4.9:** Workflow of the Offline Mode: from motion recording to DAW-based stem mapping and final robot playback.

**Step 1: Multi-modal data acquisition and filtering.** The offline workflow

begins with data logging. As the Unitree Go2 executes a predefined choreography, a comprehensive set of proprioceptive data is recorded from the DDS low-state stream. The logged dataset includes foot force readings, IMU orientation and angular velocity, and the 12 motor feedback channels corresponding to the hip, thigh, and calf joints of the four legs.

This step is automatic. A recording script stores the sensor values together with timestamps relative to the beginning of the dance. After recording, the signals are processed offline. The fourth-order Butterworth low-pass filter described in Section 4.1.3 is applied to suppress high-frequency mechanical noise. Because this stage is not constrained by online latency, the stronger filter can be used even though it introduces a measurable group delay.

The extracted motion events are represented as timestamped markers. Body-motion markers are derived primarily from the IMU roll, pitch, and yaw motion, while leg-motion markers are derived from filtered motor angular velocity signals. The filter delay and any known processing offsets are compensated before these markers are transferred to the music engineering stage.

**Step 2: Music engineering and stem mapping.** The music engineering stage converts the extracted motion markers into a coherent musical arrangement. This stage is manually supervised in a Digital Audio Workstation (DAW), while the motion-marker extraction is automatic. In other words, the software provides the timing and motion-feature information, but the artistic decisions—such as which stem should represent a roll, a yaw motion, or a calf extension—are selected and fine-tuned manually.

The decomposition process is performed as follows:

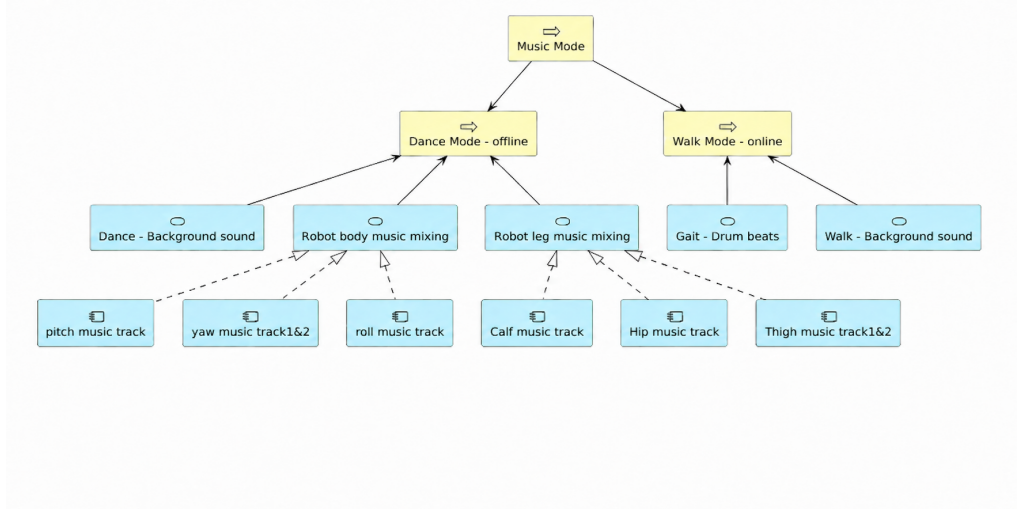
1. A source composition with a fixed tempo and suitable musical structure is selected as the base material.
2. The composition is separated into stems, such as background harmony, percussion, bass, strings, and melodic accents. If original multitrack stems are available, they are imported directly; otherwise, the stems are prepared manually or with DAW-assisted separation tools.
3. The filtered robot motion signals are converted into timestamped marker tracks. Each marker contains the motion type, the timestamp, and the relative strength of the detected event.
4. The marker tracks are imported into the DAW timeline and aligned to the same time origin as the dance recording.
5. Motion features are assigned to musical stems. For example, body pitch, yaw, and roll motions can be mapped to melodic or synthesizer stems, while hip, thigh, and calf events can be mapped to rhythmic or percussive accents.
6. The stem gain, onset position, and fade envelopes are adjusted so that the triggered musical events follow the robot movement without masking the background track.

Table 4.1 summarizes the mapping strategy used in the offline dance arrangement.

**Table 4.1:** Offline motion-to-stem mapping strategy.

| Motion feature      | Sensor source                      | Musical element        |
|---------------------|------------------------------------|------------------------|
| Body roll           | IMU angular velocity / Euler angle | Roll music track       |
| Body pitch          | IMU angular velocity / Euler angle | Pitch music track      |
| Body yaw            | IMU angular velocity / Euler angle | Yaw music tracks       |
| Hip motion          | Motor angular velocity             | Hip music track        |
| Thigh motion        | Motor angular velocity             | Thigh music tracks     |
| Calf motion         | Motor angular velocity             | Calf music track       |
| Full dance duration | DAW timeline                       | Background music track |

Because the stems are derived from the same source composition or arranged within the same tempo grid, the layered tracks remain harmonically coherent. The mapping between extracted motion features and audio elements is illustrated in Figure 4.10. In this diagram, the upper nodes represent the offline music-mode selection and the lower nodes represent the stem tracks used in the DAW arrangement.

**Figure 4.10:** Motion-to-stem mapping used for the FIDO offline dance arrangement.

**Step 3: Rendering and embedded deployment.** After the DAW mapping is adjusted and the auditory result is satisfactory, the final mix is rendered into a static audio file, such as `.wav` or `.mp3`. This rendering step is manual, because it is part of the DAW production workflow. The runtime playback step, however, is automatic.

The rendered file is transferred to the onboard Jetson computer used by the Unitree Go2 extension dock. A lightweight listener monitors the robot’s `rt/sportmodestate` topic. When the listener detects the state transition corresponding to the predefined dance action, it starts playback of the pre-rendered audio file. Since the computationally expensive filtering, musical editing, and rendering have already been

completed offline, the robot only needs to play a static file during execution. This allows complex choreographies to be performed with no online synthesis overhead.

### 4.3.3 Support for Multiple Dance Modes

The Unitree Go2 provides more than one predefined dance routine. The proposed offline workflow can be applied to each routine because the mapping is based on timestamps relative to the start of the selected dance. The same source music track and stem library can be reused across different dance modes, but each dance requires its own event timeline and final rendered audio file. Reusing the same final rendered file without modification would only work if the two dance routines had identical timing and motion-event structure, which is not generally the case.

Therefore, for two dance modes, the practical workflow is to keep the same musical material and DAW project template, record each dance separately, extract its own motion markers, and render one synchronized audio file per dance mode. At run-time, the `rt/sportmodestate` listener selects the corresponding pre-rendered file according to the detected dance action.

## 4.4 Online Mode

### 4.4.1 Real-Time Signal Processing Simplification

Unlike the Offline Mode, the Online Mode must react to walking motions while the robot is moving. The online system therefore reduces computational overhead by using only the foot force channels required for gait-event detection. In the C++ implementation, the node subscribes directly to the Unitree SDK2 DDS `rt/lowstate` topic and processes the front-left and front-right foot force values in the subscriber callback.

To reduce latency and CPU load, the fourth-order Butterworth filter is replaced by a first-order Infinite Impulse Response (IIR) low-pass filter, implemented as an Exponential Moving Average (EMA):

$$y_i[n] = \alpha x_i[n] + (1 - \alpha)y_i[n - 1], \quad (4.5)$$

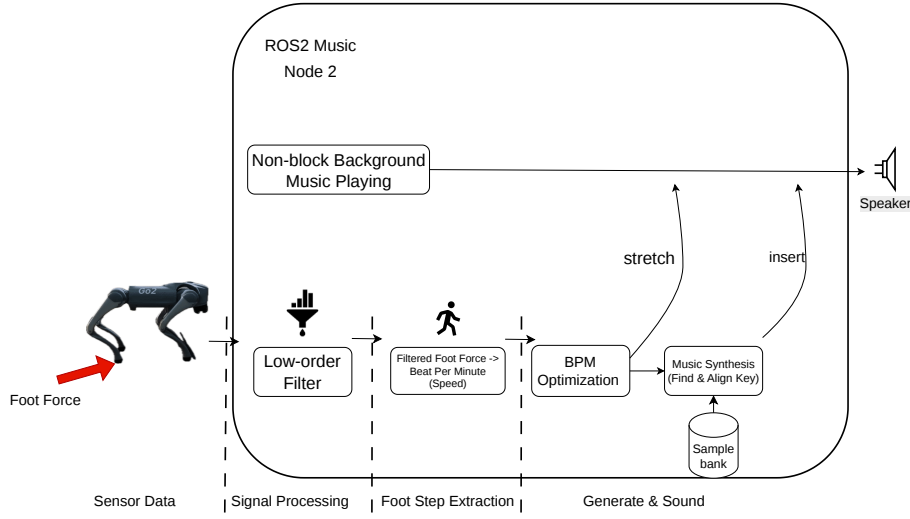
where  $x_i[n]$  is the raw foot force value of leg  $i$ ,  $y_i[n]$  is the filtered value, and  $\alpha = 0.4$  is used in the implementation. This filter has negligible computational cost and requires only one stored previous value per foot force channel.

### 4.4.2 Online Motion-to-Music Algorithm

The online sonification algorithm consists of four stages: gait-event detection, BPM estimation, tempo smoothing, and beat-synchronous audio injection. The overall architecture is illustrated in Figure 4.11.

For each incoming `rt/lowstate` message, the algorithm proceeds as follows:

1. **Filter foot force:** Apply the EMA filter to the front-left and front-right foot force readings.



**Figure 4.11:** Architecture and mapping logic of the Online Mode. Diagram created by the authors.

2. **Detect a gait event:** A front foot is considered unloaded when its filtered force is below the threshold  $F_{th} = 9$ . If this condition persists for more than 10 consecutive samples and the same foot has not already triggered in the current swing phase, a gait event is emitted. The trigger state is reset only after the foot force rises above the threshold for a small tolerance window, preventing repeated triggers due to noise.
3. **Estimate walking BPM:** When a new gait event is detected, the interval to the previous valid event is measured. Intervals shorter than 0.3 s are rejected as noise, and intervals longer than 2.0 s reset the history. The BPM estimate is computed from the average of the latest four valid intervals:

$$B_{walk} = \frac{60}{\frac{1}{M} \sum_{k=1}^M \Delta t_k}, \quad M \leq 4. \quad (4.6)$$

4. **Normalize the music tempo:** To avoid unrealistically fast playback, the walking BPM is folded toward the base music tempo  $B_0 = 40$  BPM:

$$m = \max \left( 1, \text{round} \left( \frac{B_{walk}}{B_0} \right) \right), \quad B_{target} = \frac{B_{walk}}{m}. \quad (4.7)$$

5. **Update the audio engine:** The target BPM is sent to the audio engine, and a foreground cello sample is scheduled on the current or next compatible beat of the background music.

This algorithm prevents the audio tempo from following every instantaneous fluctuation in the foot force signal. Instead, it uses threshold persistence, interval validation, short-window averaging, and tempo normalization to produce a stable target BPM.

### Tempo Transitions and Beat Alignment

The background music (BGM) is played continuously as the acoustic foundation of the online mode. Its original tempo is 40 BPM. The C++ audio engine maintains a beat grid using the current sample rate, the current BPM, and the accumulated audio frame count. The duration of one beat is:

$$N_{\text{beat}} = \frac{F_{\text{audio}} \cdot 60}{B}, \quad (4.8)$$

where  $F_{\text{audio}} = 44100$  Hz and  $B$  is the current music BPM.

Tempo changes are not applied abruptly inside an arbitrary audio buffer. Instead, the audio engine updates the playback rate on beat boundaries. A `BpmSmoother` stores the current BPM and the target BPM. When a new target is received, the smoother changes the current BPM by at most 10 BPM per beat and ignores changes smaller than a 1 BPM hysteresis threshold. The playback rate passed to `signalsmith-stretch` is then:

$$r = \frac{B}{B_0}, \quad (4.9)$$

where  $B_0 = 40$  BPM is the original tempo of the background track. This preserves pitch while allowing the BGM to accelerate or decelerate smoothly with the robot's sustained walking speed.

### Dynamic Instrumental Injection and Buffer Mixing

Beyond scaling the BGM tempo, the system injects short foreground musical responses when gait events are detected. The implementation uses eight cello samples, indexed according to the current beat position in the BGM's harmonic cycle. This is the practical form of harmonic matching used in the online system: rather than estimating a new musical key from audio in real time, the engine tracks the current beat index of the known background loop and selects the cello sample prepared for that harmonic position.

When a gait event is detected, the audio engine calls a beat-selection routine. If there is enough time left in the current beat for the cello sample to start with a safety margin, the sample is assigned to the current beat. Otherwise, it is assigned to the next beat. If the requested beat has already passed by the time the audio callback processes the trigger, the sample is dropped rather than played in a harmonically incorrect position.

The final audio output is generated inside the SDL audio callback. For each callback, the output buffer is first cleared to zero. The engine then activates any pending tracks whose scheduled start frame lies inside the current audio block. Active tracks include the time-stretched looping BGM and any foreground cello clips. Their stereo PCM samples are summed into the output buffer frame by frame. After mixing, the buffer is clamped to the valid floating-point audio range  $[-1, 1]$  and returned to SDL for playback. This buffer-level mixing design allows footstep-triggered musical events to be combined with the continuous BGM without blocking the high-priority audio callback on disk I/O or DAW-level processing.

## 4.5 Real-Time Integration Strategy

The deployed online sonification system runs on the Unitree Go2 extension dock with an onboard Jetson computer running Ubuntu 20.04 (L4T). The final design separates sensing and audio rendering into two time-critical execution contexts:

- **Unitree DDS low-state callback:** Receives `rt/lowstate`, filters the front-foot force signals using an EMA, detects gait events, estimates the target BPM, and queues foreground audio triggers.
- **SDL audio callback:** Runs whenever the audio device requests a new buffer. It updates the beat grid, smooths tempo changes, activates pending tracks, mixes their samples into the output buffer, and returns the completed buffer to the audio device.

This design avoids placing the audio-critical path inside a general ROS 2 executor. The broader robot system still uses ROS 2, but the online music node subscribes directly to the Unitree DDS low-state topic for the high-frequency foot force loop. The offline dance playback listener is separate: it monitors `rt/sportmodestate` only to choose and start a pre-rendered dance audio file.

### 4.5.1 Thread-Safe Communication Between Sensing and Audio

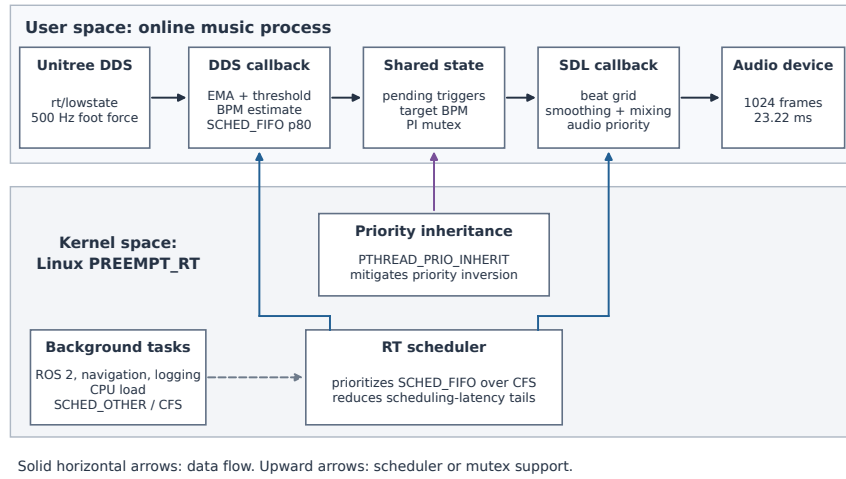
The DDS callback and the SDL audio callback share a small amount of state, including the target BPM and the queue of pending footstep-triggered clips. A mutex is therefore required to avoid race conditions. However, an ordinary `std::mutex` can cause priority inversion: if a lower-priority sensing thread holds the lock while the high-priority audio callback is ready to run, the audio thread may be blocked long enough to miss its buffer deadline.

To reduce this risk, the implementation uses a POSIX mutex configured with the `PTHREAD_PRIO_INHERIT` protocol. With priority inheritance, a thread holding the shared audio lock temporarily inherits the priority of the blocked higher-priority thread. This bounds the duration of the inversion and helps the lock holder finish its critical section quickly. In addition, expensive work such as WAV loading, time-stretch stream construction, and file logging is kept outside the real-time hot path. Runtime logs are buffered in memory and flushed to disk only after shutdown.

### 4.5.2 OS-Level Thread Prioritization

The same binary supports both baseline and real-time configurations through environment variables. The POSIX scheduling policy of the main DDS processing thread is selected using `MUSIC_CAGENT_SCHED_POLICY`. For example, the thread can run under `SCHED_FIFO` at priority 80.

The SDL audio callback priority is configured separately using `MUSIC_CAGENT_AUDIO_PRIORITY`. When real-time scheduling is enabled, memory locking can also be requested to reduce latency caused by page faults.



**Figure 4.12:** OS-level integration utilizing the Linux RT scheduler patch. The Linux scheduling mechanisms are described in [12, 34, 36]. The `PREEMPT_RT` layer represents the real-time experimental configuration.

The Raspberry Pi platform was used as a controlled evaluation platform for comparing the generic Linux scheduler against a `PREEMPT_RT` configuration. The final robot-side deployment, however, runs on the Jetson computer in the Unitree Go2 extension dock. Therefore, Raspberry Pi measurements are interpreted as scheduler-comparison evidence, while the Jetson robot logs are used to validate the latency of the deployed online sonification pipeline.

Figure 4.12 illustrates the integration of the online music process with kernel-level scheduling and synchronization mechanisms in the `PREEMPT_RT` experimental configuration. By combining a lightweight DDS callback, an SDL hardware audio callback, priority-inheritance locking, and optional `SCHED_FIFO` scheduling, the system prioritizes the critical path of filling the next audio buffer before the sound card underruns. This architecture supports responsive online gait sonification while keeping the more complex DAW-based music production in the offline dance workflow.

# 5

## Results

This chapter presents the experimental results of the proposed motion-driven sonification pipeline in both operating modes. The *Offline Dance Mode* is evaluated in terms of signal quality, motion-feature extraction, and the mapping of detected body and leg movements to pre-rendered musical stems. The *Online Walking Mode* is evaluated in terms of real-time footstep detection, communication reliability, latency, scheduling determinism, and onboard performance on the Unitree Go2’s NVIDIA Jetson computer.

The chapter is organized as follows. Section 5.1 reports the offline dance-mode signal processing and motion extraction results. Section 5.2 presents the online walking-mode gait extraction and audio-triggering behavior. Section 5.3 then evaluates the real-time implementation under controlled scheduler and CPU-load conditions, and Section 5.4 reports the final onboard deployment results collected from the FIDO robot.

### 5.1 Offline Dance Mode Results

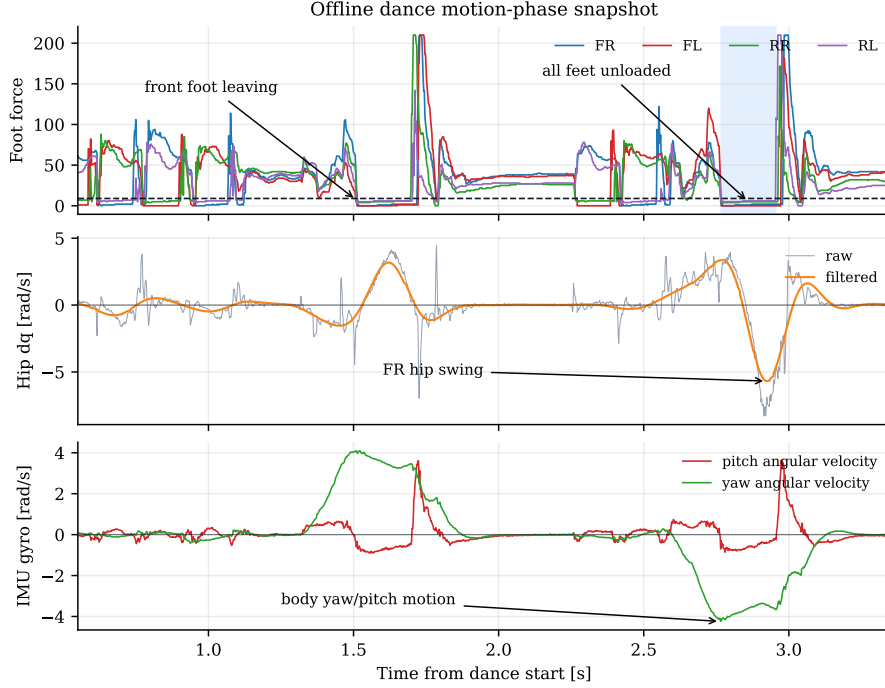
The offline mode uses recorded multi-modal robot data to extract motion events before the final audio is rendered. Because the audio is produced after data collection, this mode is not constrained by hard runtime deadlines. Its evaluation therefore focuses on whether the recorded signals contain clear and repeatable movement features that can be mapped to musically meaningful audio stems.

#### 5.1.1 Dance Motion Snapshot and Feature Mapping

Figure 5.1 shows a representative time-series snapshot from the *Dance1* dataset. The figure aligns the foot-force, motor-velocity, and IMU angular-velocity signals on the same time axis, so that visually identifiable robot movement phases can be related directly to the extracted signal features.

The upper plot shows the ground reaction force measured at each leg. Low force values indicate that a foot is unloaded or leaving the ground, while sharp force increases correspond to renewed contact. The middle plot shows the front-right hip angular velocity before and after low-pass filtering; the filtered curve preserves the dominant hip-swing motion while suppressing high-frequency motor-controller noise. The lower plot shows IMU angular velocity, where larger pitch and yaw excursions correspond to whole-body motion during the dance sequence. These

examples illustrate how the offline pipeline links observable movement phases, such as foot unloading, hip swing, and body rotation, to the sensor features used for music mapping.



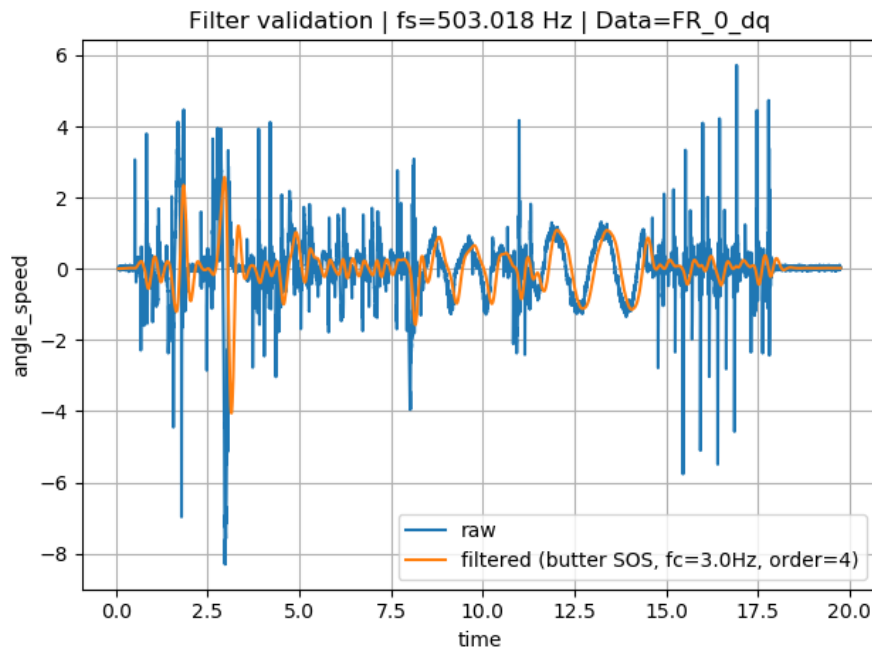
**Figure 5.1:** Representative offline dance-mode motion snapshot from the *Dance1* dataset. Foot-force, hip motor velocity, and IMU angular-velocity signals are aligned on the same time axis to show how movement phases are mapped to extractable signal features.

### 5.1.2 Butterworth Low-Pass Filter Performance

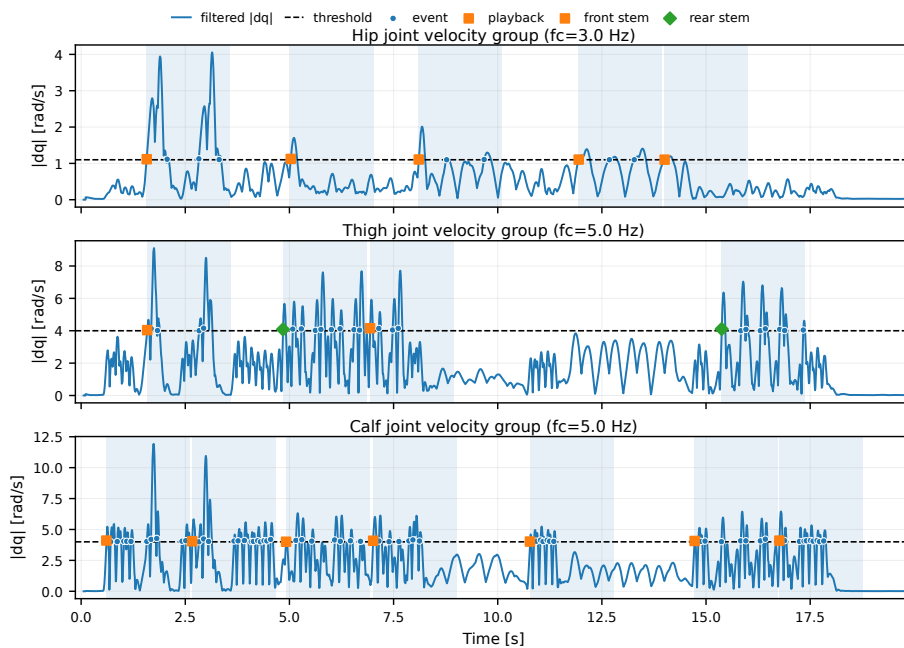
To extract meaningful kinematic triggers from the offline dance recordings, high-frequency noise in the proprioceptive motor signals must be reduced. As demonstrated in Figure 5.2, the dominant motion of the robot’s hip is preserved after applying a fourth-order Butterworth low-pass filter with a cutoff frequency of 3 Hz. The filtered signal, highlighted in orange, smooths the raw mechanical jitter while maintaining the main phase and amplitude characteristics of the leg’s swinging motion.

### 5.1.3 Motion Extraction from Motor Data

The motor joint velocity ( $dq$ ) signals are processed using the fourth-order Butterworth low-pass filter described above. A motion event is generated when the filtered signal crosses an empirically selected threshold. To avoid repeated triggering from a single physical movement, each accepted event opens a fixed-duration playback interval, during which additional triggers of the same motion type are ignored. The shaded regions in Figure 5.3 represent these active playback intervals.



**Figure 5.2:** Raw and filtered signals of the FR (Front Right) hip motor velocity, demonstrating noise suppression while preserving the dominant leg-swinging motion.

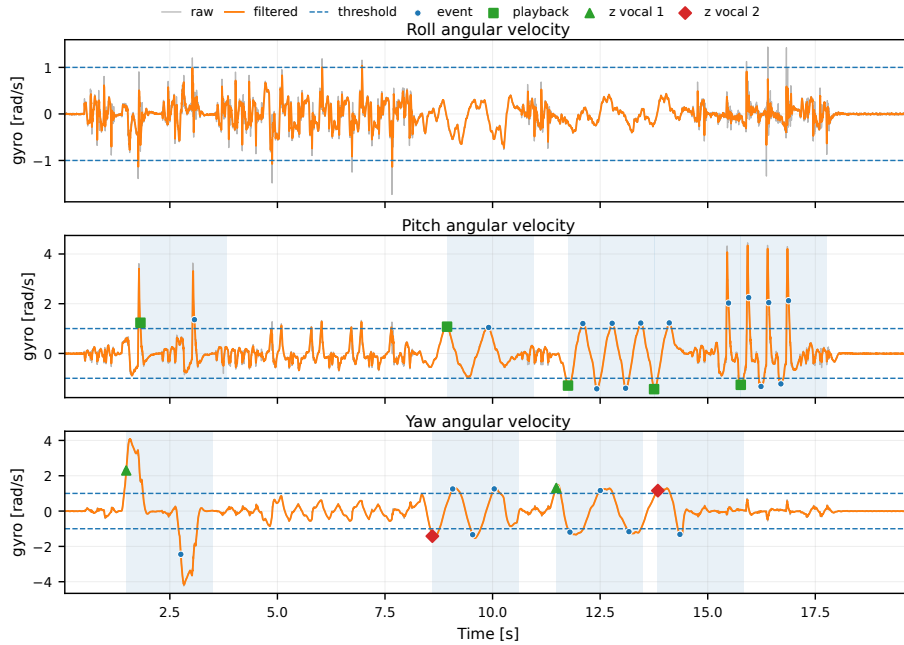


**Figure 5.3:** Motion extraction logic applied to motor joint velocity data. Shaded areas represent active sonification intervals triggered by valid threshold crossings.

#### 5.1.4 Motion Extraction from IMU Data

The IMU angular velocity signals have a lower baseline noise level than the motor velocity estimates. Consequently, the IMU channels can be smoothed with a lighter first-order low-pass filter while still retaining the body-motion features re-

quired for sonification. Figure 5.4 shows that torso-level motions, such as pitch and yaw changes, can be detected using the same threshold-and-lockout strategy used for the motor channels.



**Figure 5.4:** Motion extraction logic applied to IMU angular velocity data. The detected events represent body-motion features used for offline music mapping.

### 5.1.5 Offline Rendering Outcome

The extracted offline events are transferred to the music-production timeline as time-aligned markers. Each movement category is assigned to a corresponding audio stem, for example hip-swing motion to a rhythmic instrumental layer and body yaw or pitch motion to body-motion accent layers. Since all stems are prepared from the same musical composition, the final rendered mix remains synchronized in tempo and harmonic structure.

This procedure is performed once for each choreographed dance recording. The same musical source material can be reused across different dance modes, but the detected event timeline and the final rendered audio file are specific to each dance because the timing and type of robot movements differ between routines. Therefore, the offline result is a pre-rendered audio track that is musically synchronized to the recorded choreography and can later be triggered during robot execution without runtime music synthesis.

## 5.2 Online Walking Mode Results

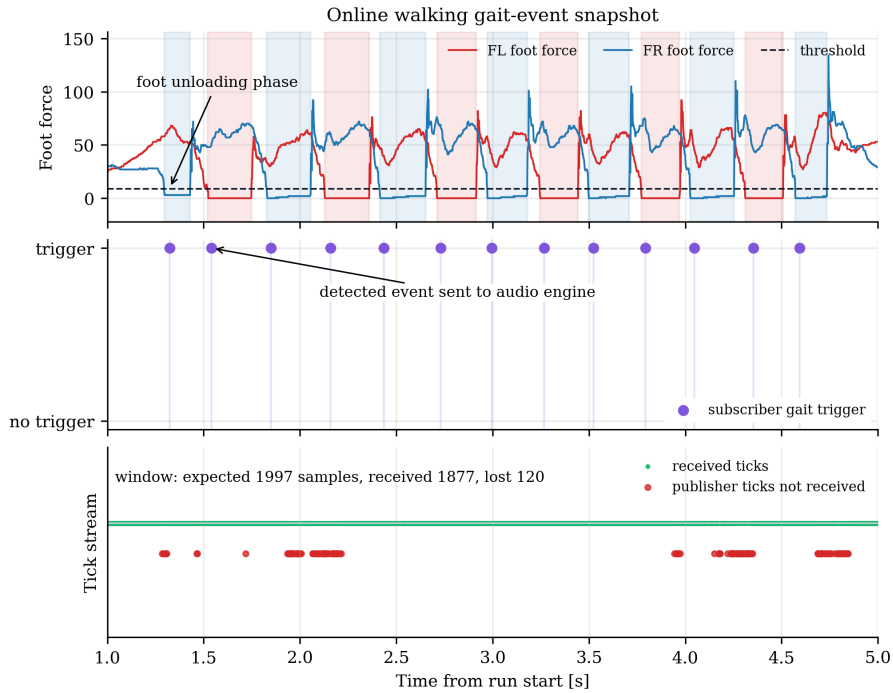
Unlike the offline dance mode, the online walking mode generates musical responses during robot locomotion. This mode therefore relies on a smaller and faster sensing pipeline: foot-force signals are filtered using an Exponential Moving Average (EMA),

gait events are detected from threshold crossings, and the audio engine schedules foreground musical samples against a continuously playing background track.

### 5.2.1 Walking Gait Snapshot and Event Detection

Figure 5.5 shows a representative online walking-mode snapshot. The upper plot displays the left and right front-foot force channels used by the online detector. Alternating low-force intervals correspond to foot unloading phases during the gait cycle. When a filtered force signal falls below the threshold and satisfies the debounce condition, the subscriber callback emits a gait trigger, shown in the middle plot. These triggers are then passed to the audio engine, where they are either mixed into the first available buffer or deferred to a musically aligned beat.

The lower plot illustrates the tick stream used to evaluate communication reliability. Green marks indicate sensor samples received by the subscriber, while red marks indicate publisher ticks that were not observed by the callback within the measurement window. This visualization links the time-series gait behavior to the sample-loss statistics reported in Section 5.3.



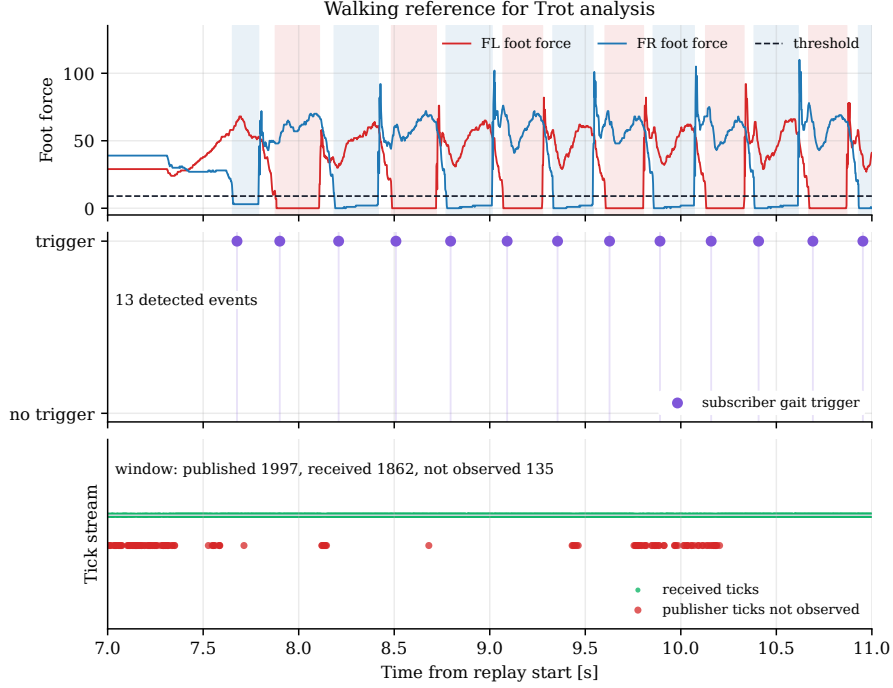
**Figure 5.5:** Representative online walking-mode gait-event snapshot. Foot-force unloading phases produce gait triggers, and the corresponding tick stream visualizes received and missed sensor samples during the same time window.

### 5.2.2 Behavior across Walking Conditions

Figure 5.6 provides a detailed view of gait-event detection during Trot walking. In the upper row, the front-left and front-right foot-force signals show alternating unloading phases and renewed force increases. The middle row marks the recorded

detections associated with these changes, illustrating how the sampled force signals are represented as discrete gait events.

The three rows thus relate the recorded force pattern to the detector output and sample reception within a common time window.



**Figure 5.6:** Representative online Trot walking gait-event snapshot obtained by replaying recorded Trot data. Foot-force unloading phases are shown alongside detected gait triggers, while the tick stream visualizes received and unobserved publisher samples during the same time window.

A separate onboard recording provides an exploratory comparison of two walking periods, provisionally labelled Free Walk (A) and Trot (B). These labels are inferred from the recorded event patterns rather than logged gait identities.

More regular detection did not correspond to a larger proportion of immediate musical responses in this onboard session. Among events with matched audio records, 58.0% in A and 50.9% in B were mixed into the first audio buffer that observed them. Gait-event regularity and playback timing therefore describe different aspects of the online Behavior. Section 5.4.5 presents the period definitions, detailed latency distributions, and implications for tempo adaptation.

### 5.3 Controlled Real-Time Performance Evaluation

This section evaluates the real-time performance of the online motion-to-audio synchronization system under controlled scheduler and load conditions. The purpose is not to measure the musical quality of the offline mode, which is pre-rendered, but to quantify the timing behavior of the online walking pipeline where sensing, gait detection, and audio scheduling occur during execution.

### 5.3.1 Evaluation Platform and Experimental Setup

To evaluate scheduler behavior without modifying the proprietary Unitree Go2 operating system, a Raspberry Pi 5 was used as a surrogate real-time evaluation platform. This allowed kernel-level comparisons to be performed without risking the robot’s onboard software environment. The controlled experiment focused on predictability and determinism rather than raw computational speed.

Four configurations were tested by combining two scheduler/kernel settings and two load conditions. The first scheduler setting was the default Linux *Completely Fair Scheduler* (CFS), exposed as `SCHED_OTHER`. CFS is designed to share CPU time fairly among normal processes, but it does not provide hard real-time guarantees. The second setting used a `PREEMPT_RT` kernel with the online music callback elevated to `SCHED_FIFO` priority 80, which gives the real-time audio/sensing path priority over normal background tasks. The two load conditions were idle operation and heavy CPU stress, resulting in the four configurations shown in Table 5.1.

**Table 5.1:** Configuration of the four controlled real-time experimental conditions.

| Condition     | Kernel                  | Executor policy                    | Background load         | Runs |
|---------------|-------------------------|------------------------------------|-------------------------|------|
| CFS, idle     | Generic                 | <code>SCHED_OTHER</code> (CFS)     | None                    | 12   |
| CFS, load     | Generic                 | <code>SCHED_OTHER</code> (CFS)     | CPU stress <sup>a</sup> | 12   |
| RT FIFO, idle | <code>PREEMPT_RT</code> | <code>SCHED_FIFO</code> (prio. 80) | None                    | 12   |
| RT FIFO, load | <code>PREEMPT_RT</code> | <code>SCHED_FIFO</code> (prio. 80) | CPU stress <sup>a</sup> | 12   |

<sup>a</sup> CPU stress was generated using `stress-ng --cpu 4 --timeout 130s`.

For each run, the publisher emitted 65,000 simulated walking low-state samples at approximately 500 Hz. Each sample carried a monotonically increasing tick index, allowing the subscriber to determine whether intermediate samples had been missed. The ROS 2 QoS profile was configured as `BEST_EFFORT` with a 5 ms lifespan and a 3 ms deadline. The SDL audio hardware callback used a buffer size of 1024 frames at 44.1 kHz, giving a nominal buffer period of

$$T_{\text{buf}} = \frac{1024}{44100} \approx 23.22 \text{ ms.}$$

### 5.3.2 ROS 2 Communication Reliability

The integrity of the 500 Hz foot-force stream is summarized in Table 5.2. A sample is counted as lost when the subscriber observes a gap in the publisher tick sequence. In other words, the loss percentage is computed from the difference between the expected number of published ticks and the number of ticks actually received by the callback:

$$\text{loss} = \frac{N_{\text{expected}} - N_{\text{received}}}{N_{\text{expected}}} \times 100\%.$$

This definition captures both transport-level loss and scheduler-induced callback starvation under the configured short-lifespan QoS policy.

**Table 5.2:** Sampling integrity across scheduler and load conditions.

| Condition     | Lost samples (%)         | Effective sampling period (ms) |
|---------------|--------------------------|--------------------------------|
| CFS, idle     | 10.91 [10.34, 11.99]     | 1.947 [1.940, 1.952]           |
| CFS, load     | 30.16 [29.54, 31.08]     | 2.964 [2.946, 2.967]           |
| RT FIFO, idle | 6.15 [5.05, 6.55]        | 1.940 [1.933, 1.944]           |
| RT FIFO, load | <b>8.30</b> [8.02, 8.47] | <b>1.997</b> [1.996, 1.999]    |

Values are reported as median [bootstrap 95% confidence interval] over 12 runs. Bold values highlight the real-time configuration under CPU load.

Under CPU stress, the default CFS configuration lost 30.16% of the samples, and its effective sampling period increased to 2.964 ms. By contrast, the real-time configuration using `PREEMPT_RT` and `SCHED_FIFO` reduced the sample-loss rate to 8.30% while maintaining an effective sampling period of 1.997 ms. These results show that real-time scheduling substantially improves the reliability of the high-rate sensing stream under competing CPU load.

### 5.3.3 Execution Jitter and Priority Inversion

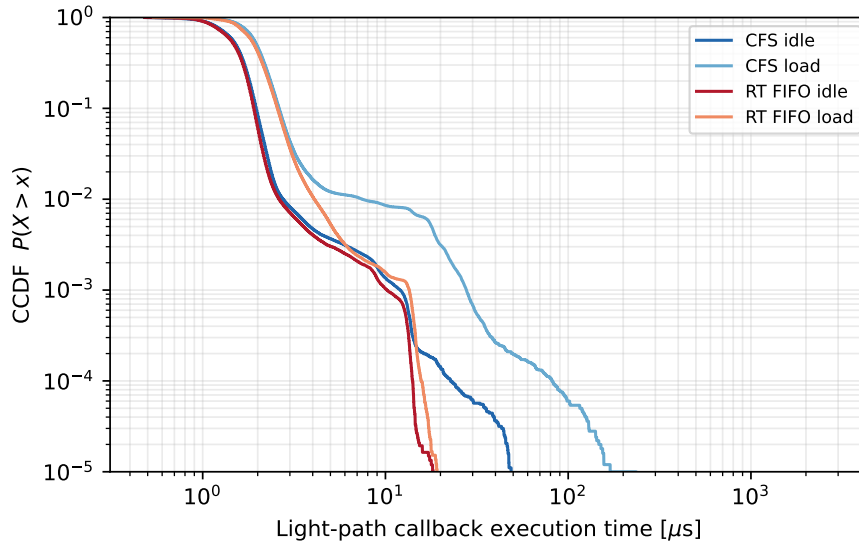
To isolate OS-level preemption, the execution time of the *light subscriber callback* was measured. This path corresponds to samples where no gait event is detected; it only updates the EMA filter and checks the threshold. Because the computational work is nearly constant, long-tail increases in execution time primarily indicate scheduler interference.

**Table 5.3:** Execution time of the light-path subscriber callback.

| Condition     | $N$     | $p_{50}$<br>( $\mu$ s) | $p_{99}$<br>( $\mu$ s) | $p_{99.9}$<br>( $\mu$ s) | Max<br>( $\mu$ s) | Max/ $p_{50}$<br>( $\times$ ) |
|---------------|---------|------------------------|------------------------|--------------------------|-------------------|-------------------------------|
| CFS, idle     | 622,060 | 1.52                   | 2.76                   | 11.96                    | 53.4              | 35                            |
| CFS, load     | 481,814 | 1.98                   | 7.46                   | 27.80                    | 2,875.0           | <b>1,451</b>                  |
| RT FIFO, idle | 667,902 | 1.48                   | 2.61                   | 10.19                    | 20.9              | 14                            |
| RT FIFO, load | 650,699 | 1.91                   | 4.07                   | 13.48                    | 24.7              | <b>13</b>                     |

The inflation factor is defined as  $\text{Max}/p_{50}$ . Larger values indicate more severe long-tail execution-time inflation. Bold values highlight the comparison between the two loaded configurations.

Table 5.3 and Figure 5.7 show that the median callback execution time is similar across all conditions, remaining close to  $2\mu$ s. The important difference appears in the tail. Under load, CFS produced a worst-case execution time of  $2,875.0\mu$ s, whereas the RT FIFO configuration bounded the worst case to  $24.7\mu$ s. Thus, the real-time configuration does not make the average computation faster; instead, it prevents rare scheduler delays from growing into large outliers.



**Figure 5.7:** Complementary Cumulative Distribution Function (CCDF) of light callback execution times. The loaded CFS curve shows a severe non-deterministic long tail.

Table 5.4 reports the latency of the audio-facing path. Handoff latency is measured from the moment a gait event is detected until the first SDL audio callback observes the queued trigger. Heavy callback time measures the callback path where a valid trigger is processed and submitted to the audio engine.

**Table 5.4:** Latency of the audio-facing paths across 12 runs.

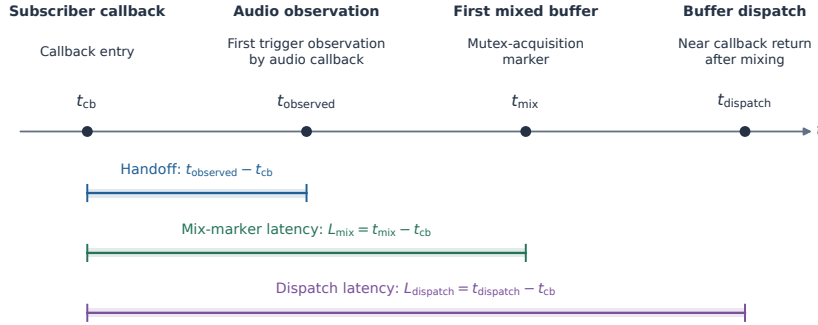
| Condition     | Handoff latency (ms) |          |            |              |                             | Heavy callback (ms) |          |       |
|---------------|----------------------|----------|------------|--------------|-----------------------------|---------------------|----------|-------|
|               | $p_{50}$             | $p_{99}$ | $p_{99.9}$ | Max          | $\text{Max}/T_{\text{buf}}$ | $p_{50}$            | $p_{99}$ | Max   |
| CFS, idle     | 12.77                | 34.74    | 42.18      | 42.54        | 1.83                        | 0.025               | 13.59    | 21.93 |
| CFS, load     | 13.97                | 38.95    | 45.38      | <b>49.83</b> | <b>2.15</b>                 | 0.163               | 23.01    | 32.99 |
| RT FIFO, idle | 12.59                | 29.99    | 31.62      | 32.44        | 1.40                        | 0.026               | 13.92    | 16.14 |
| RT FIFO, load | 13.93                | 34.81    | 40.21      | <b>42.96</b> | <b>1.85</b>                 | 0.034               | 14.98    | 16.86 |

Handoff latency is measured from gait-event detection until the audio thread observes the trigger. The SDL buffer period is  $T_{\text{buf}} = 23.22$  ms. Bold values highlight the comparison between the two loaded configurations.

The loaded CFS condition reached a maximum handoff latency of 49.83 ms, corresponding to 2.15 audio-buffer periods. The RT FIFO configuration reduced the maximum handoff latency under the same stress condition to 42.96 ms, or 1.85 buffer periods. This reduction is important because the sensing callback and the audio callback share data across thread boundaries; when the sensing thread is delayed behind lower-priority work, the audio thread may not observe the trigger until a later buffer.

### 5.3.4 End-to-End Latency and Musical Quantization

End-to-end latency is measured from the detected physical gait event to the dispatch of the mixed audio buffer. Figure 5.8 summarizes the timing points used in the measurement. The system records the trigger time in the sensing callback, the time at which the audio callback first observes the queued trigger, and the time at which the selected audio sample is mixed into an output buffer.



**Figure 5.8:** Software timing markers and latency intervals in the online footstep-to-audio pipeline. The coloured spans show handoff, mix-marker, and dispatch latencies measured from subscriber callback entry. The timeline is schematic and not to scale.

As shown in Table 5.5 and Figure 5.9, the end-to-end latency distribution is bimodal. This does not indicate two different computational speeds. Instead, it reflects two different scheduling decisions made by the music engine. *Immediate* events are mixed into the first available audio buffer after the trigger is observed. Their latency is therefore dominated by system-induced delay: sensing callback timing, thread handoff, and the fixed phase of the SDL buffer. *Deferred* events are valid gait events that are intentionally delayed by the beat-alignment algorithm so that the foreground musical sample starts on a musically compatible beat of the background track.

The immediate population remains below 50 ms in all controlled conditions, including the loaded CFS condition. This confirms that the core sensing-to-audio path is fast enough for perceptual synchrony when an event is allowed to play immediately. The deferred population, with median delays above 500 ms, is instead dominated by musical quantization. These delays are intentionally introduced by the algorithm to maintain rhythmic coherence with the background track and should therefore not be interpreted as failures of the operating system scheduler.

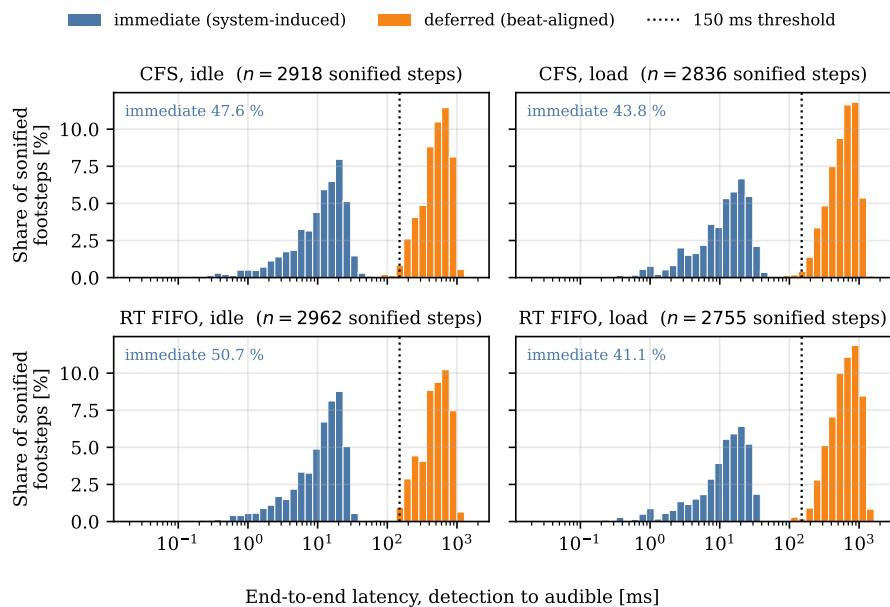
### 5.3.5 Run-Level Synthesis and Overall Determinism

Table 5.6 aggregates the run-level metrics and reports the ratio  $R$  between the CFS and RT configurations. Figures 5.10 and 5.11 visualize the same comparison at the run level.

**Table 5.5:** End-to-end latency by audio-alignment behavior.

| Condition     | Steps | Immediate |               |               |          | Deferred  |               | All steps<br>$\leq 150$ ms (%) |
|---------------|-------|-----------|---------------|---------------|----------|-----------|---------------|--------------------------------|
|               |       | Share (%) | $p_{50}$ (ms) | $p_{99}$ (ms) | Max (ms) | Share (%) | $p_{50}$ (ms) |                                |
| CFS, idle     | 2,918 | 47.6      | 12.89         | 34.24         | 42.54    | 52.4      | 531.6         | 48.5                           |
| CFS, load     | 2,836 | 43.8      | 12.99         | 37.58         | 49.83    | 56.2      | 613.4         | 44.5                           |
| RT FIFO, idle | 2,962 | 50.7      | 12.64         | 30.01         | 32.44    | 49.3      | 523.5         | 51.3                           |
| RT FIFO, load | 2,755 | 41.1      | 13.82         | 33.62         | 42.96    | 58.9      | 645.3         | 41.7                           |

Steps are sonified gait events pooled over 12 runs. Immediate events are mixed into the first available audio buffer, whereas deferred events wait for a compatible musical beat. The final column reports the percentage of all sonified steps rendered within 150 ms.

**Figure 5.9:** Distribution of the total end-to-end latency, showing the separation between immediate system-induced latency and beat-aligned deferred latency.

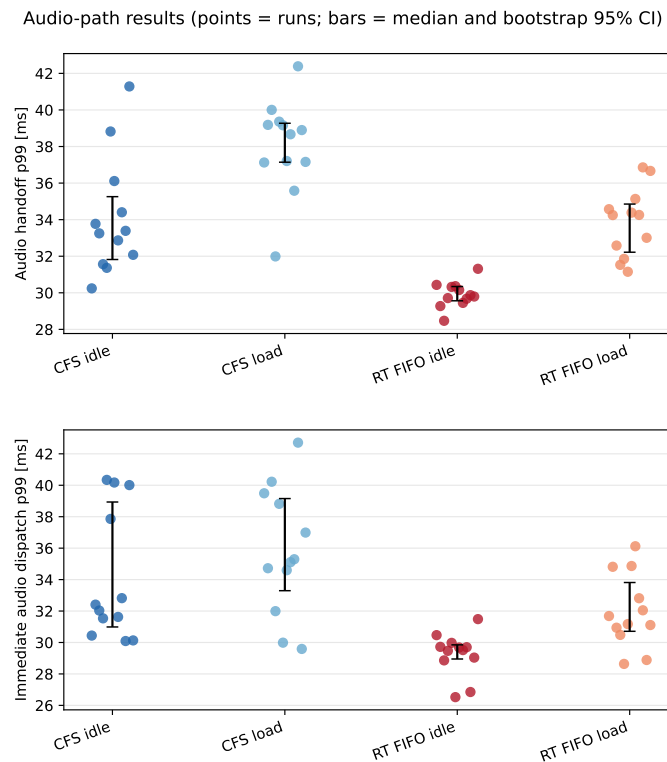
**Table 5.6:** Run-level comparison of CFS against PREEMPT\_RT with SCHED\_FIFO.  $R$  is the CFS/RT ratio. For metrics where smaller is better,  $R > 1$  favors the RT setup.

| Metric                                      | Unit    | Idle  |       |               | Under load |       |                                |
|---------------------------------------------|---------|-------|-------|---------------|------------|-------|--------------------------------|
|                                             |         | CFS   | RT    | $R$           | CFS        | RT    | $R$                            |
| <i>Sensing integrity</i>                    |         |       |       |               |            |       |                                |
| Lost samples                                | %       | 10.91 | 6.15  | $1.77\times$  | 30.16      | 8.30  | <b><math>3.63\times</math></b> |
| Sampling period p50                         | ms      | 1.947 | 1.940 | $1.003\times$ | 2.964      | 1.997 | $1.48\times$                   |
| <i>CPU scheduling (light callback)</i>      |         |       |       |               |            |       |                                |
| p50                                         | $\mu$ s | 1.52  | 1.50  | $1.012\times$ | 1.98       | 1.92  | $1.034\times$                  |
| p99.9                                       | $\mu$ s | 11.85 | 10.15 | $1.17\times$  | 27.52      | 13.50 | $2.04\times$                   |
| Max                                         | $\mu$ s | 47.3  | 18.2  | $2.60\times$  | 155.6      | 19.0  | <b><math>8.17\times</math></b> |
| <i>Priority inversion (heavy callback)</i>  |         |       |       |               |            |       |                                |
| p99                                         | ms      | 13.09 | 13.68 | $0.957\times$ | 21.62      | 14.64 | $1.48\times$                   |
| Max                                         | ms      | 19.43 | 15.24 | $1.27\times$  | 28.18      | 15.80 | <b><math>1.78\times</math></b> |
| <i>Audio handoff</i>                        |         |       |       |               |            |       |                                |
| p50                                         | ms      | 12.95 | 12.59 | $1.029\times$ | 13.93      | 13.92 | $1.000\times$                  |
| p99                                         | ms      | 33.32 | 29.83 | $1.12\times$  | 38.78      | 34.25 | $1.13\times$                   |
| Max                                         | ms      | 41.57 | 31.01 | $1.34\times$  | 43.05      | 38.08 | $1.13\times$                   |
| <i>Perceptual latency (immediate steps)</i> |         |       |       |               |            |       |                                |
| p50                                         | ms      | 12.80 | 12.87 | $0.995\times$ | 13.26      | 13.85 | $0.958\times$                  |
| Max                                         | ms      | 35.36 | 30.47 | $1.16\times$  | 39.14      | 33.98 | $1.15\times$                   |
| Immediate share                             | %       | 47.2  | 53.1  | $0.89\times$  | 43.3       | 42.8  | $1.010\times$                  |
| Steps $\leq 150$ ms                         | %       | 47.9  | 54.1  | $0.88\times$  | 44.3       | 43.0  | $1.029\times$                  |

The run-level results show that real-time scheduling mainly improves the tails of the distributions. The median values are close between CFS and RT, indicating that the algorithm itself is lightweight. Under load, however, RT scheduling improves sensing integrity by a factor of 3.63 and reduces maximum light-callback inflation by a factor of 8.17. This is the relevant result for interactive audio: the system must not merely be fast on average, but must avoid rare delays that cause missed sensing updates or late audio-buffer insertion.

## 5.4 Onboard Real-World Deployment (Unitree Go2 Jetson)

The controlled Raspberry Pi experiment isolates the effect of OS scheduling, but the final system runs on the FIDO robot’s onboard NVIDIA Jetson computer. This section reports the real-world deployment results collected during live robot locomotion. The goal is to verify whether the optimized C++ online music node remains fast enough on the robot’s actual onboard computer, even without replacing the proprietary Unitree operating system with a custom PREEMPT\_RT kernel.



**Figure 5.10:** Run-level summary of audio handoff and perceptual latency metrics.

### 5.4.1 Onboard Jetson Log Dataset

The analysis uses telemetry logs generated by the deployed C++ online music node. Three synchronized log streams were recorded: subscriber callback timing, SDL audio callback timing, and footstep-to-audio end-to-end latency. All timing values were recorded using monotonic nanosecond timestamps. The resulting dataset is summarized in Table 5.7.

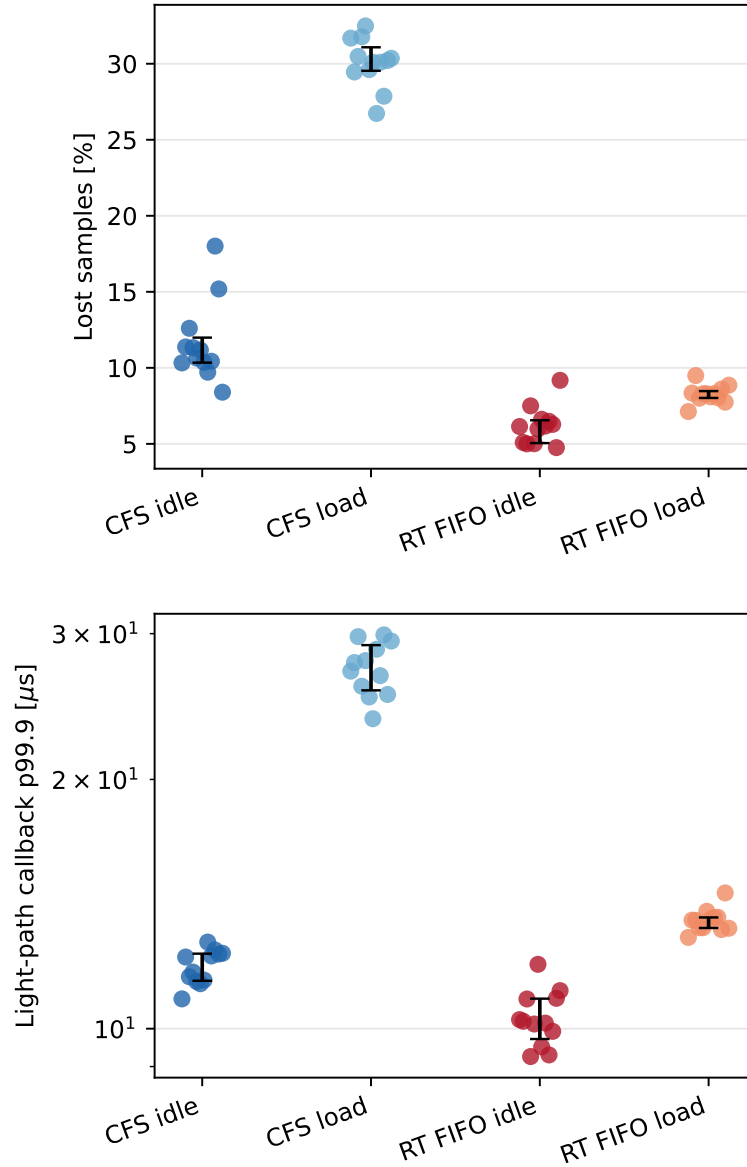
**Table 5.7:** Robot-log dataset used for the onboard Jetson latency analysis during live locomotion.

| Quantity                        | Value     |
|---------------------------------|-----------|
| Subscriber callback samples     | 522,591   |
| Detected foot-contact samples   | 1,409     |
| Stored low-state trigger rows   | 1,409     |
| Audio callback samples          | 45,036    |
| Valid footstep-to-audio samples | 1,315     |
| Measurement duration            | 1,045.3 s |
| Median subscriber period        | 1.999 ms  |

### 5.4.2 Latency Definitions and Algorithmic Separation

Four timing quantities are reported for the onboard deployment. The *light subscriber callback* corresponds to low-state samples where no foot contact is detected and only

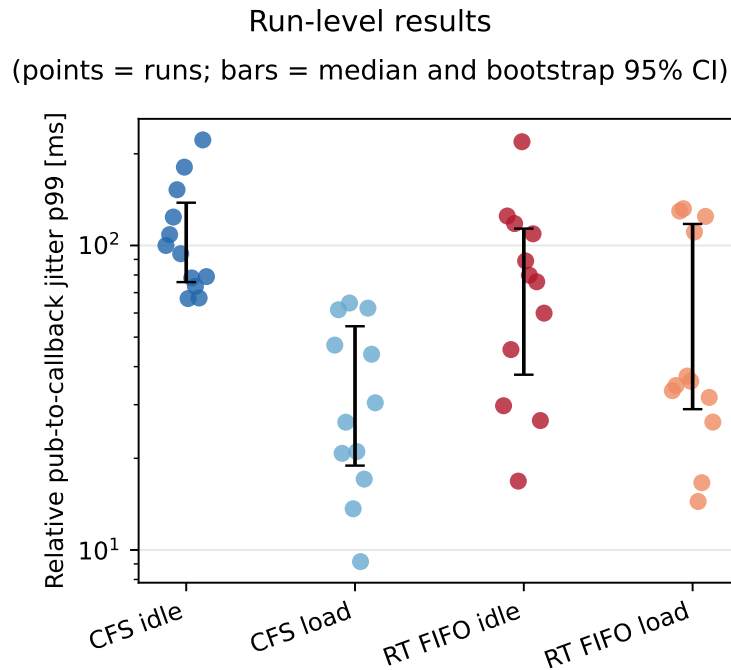
Run-level results  
(points = runs; bars = median and bootstrap 95% CI)



**Figure 5.11:** Run-level summary showing the reduction of system-induced timing outliers under the `PREEMPT_RT` kernel.

filtering is performed. The *music-trigger callback* corresponds to samples where a foot-contact event is detected and submitted to the audio queue. *Handoff latency* is the time from foot-contact detection until the first audio callback observes the trigger. *End-to-end latency* is the time from detected foot contact to the dispatched mixed audio buffer.

As in the controlled experiment, the end-to-end samples are separated into immediate and deferred events. Immediate events measure the true system-induced response time of the robot. Deferred events include an intentional waiting time in-



**Figure 5.11:** Run-level summary showing the reduction of system-induced timing outliers under the PREEMPT\_RT kernel (continued).

roduced by the beat-alignment algorithm. This distinction is essential because the deferred tail reflects musical quantization rather than computational overload.

### 5.4.3 Onboard Latency Results

Table 5.8 summarizes the primary latency statistics extracted from the robot log. The light subscriber callback had a median execution time of  $0.544\ \mu\text{s}$  and a p99 of  $2.336\ \mu\text{s}$ , confirming that the EMA filter and threshold detector consume a negligible fraction of the Jetson’s computing budget. The audio callback runtime had a median of  $9.010\ \text{ms}$  and a p99 of  $10.377\ \text{ms}$ , remaining safely below the median hardware buffer period of  $24.607\ \text{ms}$ .

**Table 5.8:** Comprehensive latency statistics derived from the onboard Jetson robot log.

| Metric                                | Unit          | $N$     | p50    | p90    | p99    | Max     |
|---------------------------------------|---------------|---------|--------|--------|--------|---------|
| Light subscriber callback             | $\mu\text{s}$ | 521,182 | 0.544  | 0.960  | 2.336  | 38.049  |
| Music-trigger callback                | ms            | 1,409   | 0.029  | 6.725  | 9.184  | 10.244  |
| Subscriber jitter $ g_i - \tilde{g} $ | ms            | 522,590 | 0.056  | 0.187  | 0.515  | 13.871  |
| Audio callback runtime                | ms            | 45,036  | 9.010  | 9.825  | 10.377 | 11.561  |
| Detection-to-audio handoff            | ms            | 1,315   | 11.554 | 23.898 | 35.483 | 39.658  |
| Immediate end-to-end                  | ms            | 716     | 10.899 | 23.614 | 35.388 | 39.658  |
| Deferred end-to-end                   | ms            | 599     | 427.6  | 724.7  | 970.8  | 1,406.8 |
| All end-to-end                        | ms            | 1,315   | 24.2   | 616.8  | 893.4  | 1,406.8 |

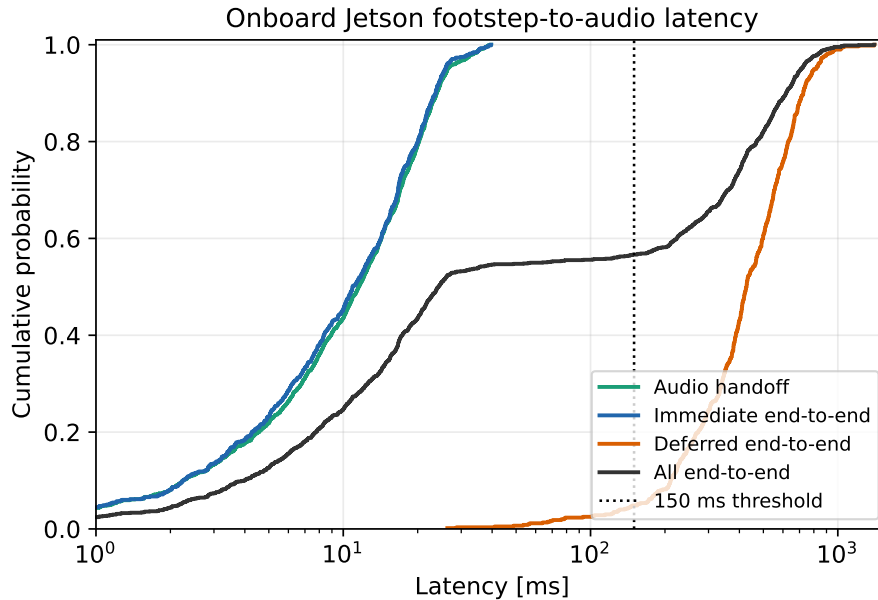
For immediate events, the footstep-to-audio latency had a median of 10.899 ms, a p99 of 35.388 ms, and a maximum of 39.658 ms, as shown in Table 5.9. These values are comfortably below the 150 ms audio-visual synchrony threshold used as the design target. Across all sonified footsteps, 56.7% were rendered below this threshold. The remaining long-latency samples are beat-aligned deferred events and therefore include intentional musical waiting time.

Figure 5.12 presents the cumulative distributions of footstep-to-audio latency measured on the onboard Jetson, including the immediate and beat-aligned deferred playback paths. Figure 5.13 summarizes the median, 99th percentile, and maximum values of the callback execution times and audio latency measurements from the robot log.

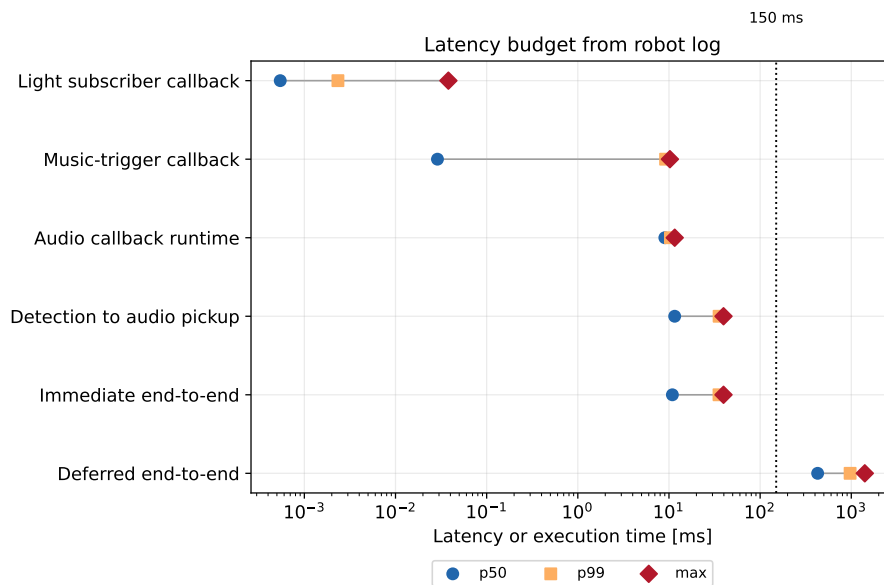
**Table 5.9:** Distribution of immediate and beat-aligned deferred footstep-to-audio latency populations on the Jetson platform.

| Playback path          | $N$   | Share (%) | p50 (ms) | p99 (ms) | Max (ms) |
|------------------------|-------|-----------|----------|----------|----------|
| Immediate              | 716   | 54.4      | 10.899   | 35.388   | 39.658   |
| Beat-aligned deferred  | 599   | 45.6      | 427.6    | 970.8    | 1,406.8  |
| All sonified footsteps | 1,315 | 100.0     | 24.2     | 893.4    | 1,406.8  |

**Note:** 56.7% of all sonified footsteps were rendered below the 150 ms perceptual synchrony threshold.



**Figure 5.12:** Cumulative distribution of the onboard Jetson footstep-to-audio latency. Immediate events reflect system-induced delay, while deferred events include intentional musical beat alignment.



**Figure 5.13:** Latency budget derived from the live robot log. The deferred end-to-end path is dominated by musical quantization delay, whereas the immediate path remains bounded below the perceptual threshold.

#### 5.4.4 Timing Jitter During On-Robot Operation

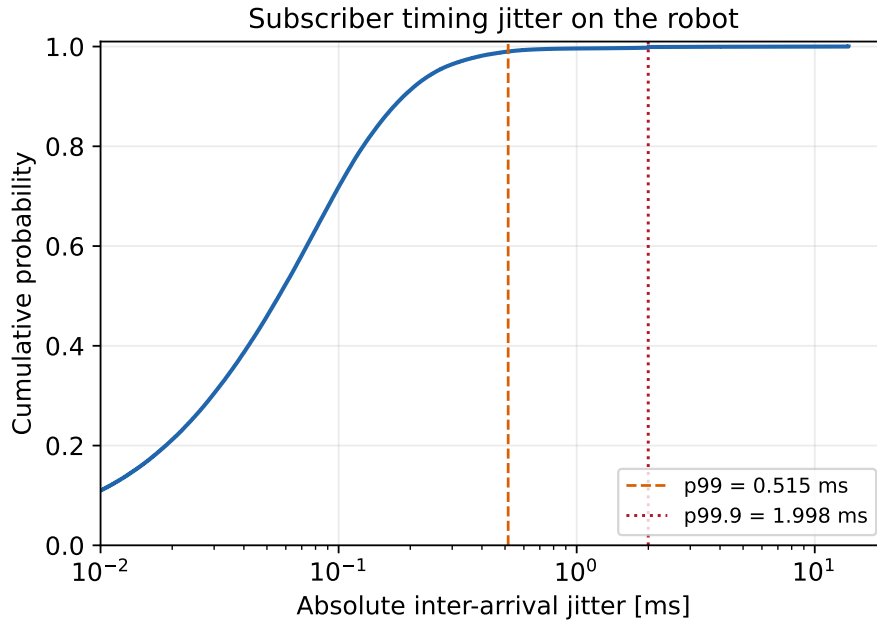
The deployed low-state subscriber processes the robot’s proprioceptive stream at approximately 500 Hz, corresponding to a nominal period of 2.0 ms. In the live robot log, the median observed callback period was 1.999 ms. As shown in Figure 5.14, the absolute inter-arrival jitter had a median of 0.056 ms and a p99 of 0.515 ms, indicating that most callbacks arrived very close to the nominal period. The maximum observed jitter was 13.871 ms, showing that occasional scheduling outliers still occur on the robot’s default Linux environment.

Despite these occasional subscriber-side outliers, the immediate end-to-end latency remained below 40 ms. This indicates that the decoupled C++ architecture successfully prevents most middleware and scheduling variation from propagating into audible response delays. The audio thread therefore remains stable enough for real-time motion-synchronous feedback during live robot walking.

#### 5.4.5 Online Behavior under Free Walk and Trot Conditions

**Scope and condition definition.** A retrospective analysis of the 1,045.347 s on-board Jetson recording was conducted to examine system behaviour under different walking conditions. Two periods exhibiting distinct detected-event patterns were selected and provisionally designated as Free Walk (period A) and Trot (period B). As gait-mode and transition metadata were not recorded, these designations represent inferred condition labels rather than verified controller modes. The analysis therefore focuses on within-session differences in event timing and software response latency, without attributing these differences solely to gait.

The candidate Free Walk period (A) spans 0–469.544 s and the candidate Trot pe-



**Figure 5.14:** Cumulative distribution of the absolute inter-arrival jitter of the low-state subscriber callback during live robot operation.

riod (B) spans 507.410–1,045.347 s relative to the first subscriber callback. A 37.866 s interval without detected events separates them; this is not evidence that the robot was physically stationary. The boundaries were selected retrospectively from the event stream, not from a recorded gait-switch command. Both periods belong to the same run already used for the aggregate onboard evaluation. They are not additional independent trials, and either period may contain pauses or other unlabelled movement.

**Event timing.** Table 5.10 and Figure 5.15 summarize the period-wise comparison. The candidate Free Walk and Trot periods contain 695 and 714 detected events, respectively. Consecutive within-period detections separated by no more than 2 s were retained for interval statistics, yielding 647 and 678 intervals; 47 and 35 longer within-period gaps were excluded. The inter-period gap was not assigned to either distribution. This rule reduces the influence of extended interruptions, but does not recover ground-truth physical step intervals.

The median detection intervals are similar: 247.96 ms in candidate Free Walk (A) and 253.91 ms in candidate Trot (B). In contrast, their interquartile ranges differ substantially: 201.90–332.23 ms versus 250.03–260.03 ms. Thus, the candidate Trot period exhibits a much more concentrated event-timing pattern, not a higher median event frequency. The same qualitative difference remains when the 2 s exclusion is removed: the quartiles are 205.90–440.20 ms in A and 250.05–262.18 ms in B.

Event-sampled front-foot forces provide an additional description of the difference. A detection was assigned a single-front low-force pattern when exactly one filtered front-foot value was at or below the detector threshold of 9. Among consecutive detections separated by at most 2 s and both having such a pattern, the low-force side

alternates in 87.8% of 645 eligible pairs in A and 94.1% of 677 pairs in B. The median lower raw front-foot value is 3 in A and 0 in B, in the recorded sensor units. The more regular, alternating pattern in B motivates treating it as the candidate Trot period; A is the complementary Free Walk candidate under the reported two-condition interpretation. These features do not identify gait uniquely. In particular, the event-sampled front-foot values cannot establish diagonal front–hind phase relations or prove that either entire period contains only one gait.

**Table 5.10:** Exploratory within-session comparison with inferred labels: Free Walk (A) and Trot (B). The assignment is not a logged or independently verified gait identity. Interval statistics exclude gaps exceeding 2 s; latency statistics include all matched events in each stated class.

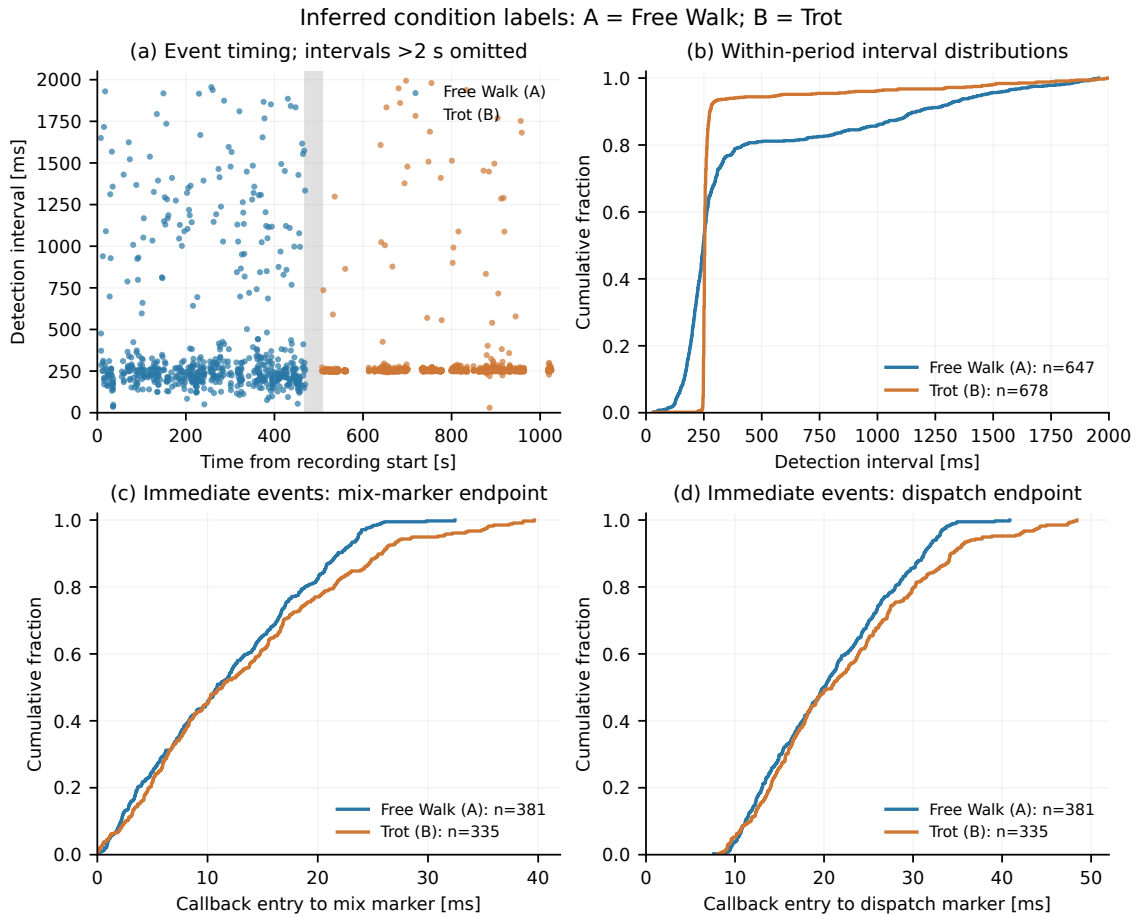
| Quantity                                  | Free Walk<br>(A; inferred) | Trot<br>(B; inferred) |
|-------------------------------------------|----------------------------|-----------------------|
| Detected events                           | 695                        | 714                   |
| Matched first-buffer records              | 657                        | 658                   |
| Unmatched detections                      | 38                         | 56                    |
| Immediate records                         | 381                        | 335                   |
| Deferred records                          | 276                        | 323                   |
| Immediate share of matched records [%]    | 58.0                       | 50.9                  |
| Retained detection intervals              | 647                        | 678                   |
| Detection interval median [ms]            | 248.0                      | 253.9                 |
| Detection interval lower quartile [ms]    | 201.9                      | 250.0                 |
| Detection interval upper quartile [ms]    | 332.2                      | 260.0                 |
| Retained intervals $\leq 300$ ms [%]      | 69.6                       | 93.2                  |
| Eligible single-front pairs               | 645                        | 677                   |
| Alternating single-front pairs [%]        | 87.8                       | 94.1                  |
| Immediate mix-marker latency median [ms]  | 10.87                      | 11.34                 |
| Immediate mix-marker latency p99 [ms]     | 25.68                      | 37.58                 |
| Immediate mix-marker latency maximum [ms] | 32.45                      | 39.66                 |
| Immediate dispatch latency median [ms]    | 19.96                      | 20.89                 |
| Immediate dispatch latency p99 [ms]       | 34.52                      | 47.40                 |
| Immediate dispatch latency maximum [ms]   | 40.87                      | 48.41                 |
| Deferred mix-marker latency median [ms]   | 411.57                     | 443.39                |

**Software response latency.** Exact trigger-timestamp matching associates 657 first-mixed-buffer records with A and 658 with B. The other 38 and 56 detections lack a matching record; their individual causes cannot be determined from these CSV files. These counts are not physical-step miss rates. Immediate events are defined by equality of the first-observation and first-mixed-buffer markers; deferred events reach their first mixed buffer later. There are 381 immediate records in A and 335 in B, representing 58.0% and 50.9% of matched records.

Two software timing endpoints are distinguished:

$$L_{\text{mix}} = t_{\text{mix}} - t_{\text{cb}}, \quad L_{\text{dispatch}} = t_{\text{dispatch}} - t_{\text{cb}}. \quad (5.1)$$

Here,  $t_{\text{cb}}$  is the entry time of the subscriber callback that detects the event. The legacy CSV field `audible_ns`, used for  $t_{\text{mix}}$ , identifies the mutex-acquisition marker



**Figure 5.15:** Exploratory Free Walk/Trot comparison from the 22 August onboard recording. Labels are inferred: A is the candidate Free Walk period and B the candidate Trot period, not logged or independently validated gait identities. (a) Detection intervals of at most 2 s; the shaded band separates A and B. Longer gaps are omitted from this panel and (b), not from latency analysis. (c–d) Immediate-event latency distributions using the first-mixed-buffer marker and the later software dispatch marker. Neither endpoint measures acoustic onset.

of the first buffer containing the triggered track; it is not a measured acoustic onset or the exact instant at which its first sample is mixed. The field `buffer_dispatch_ns` supplies a later software marker near callback return, after mixing. Neither endpoint includes a measured physical-event-to-callback delay or establishes when sound leaves the loudspeaker. Dispatch latency is calculated directly from its recorded timestamp, not by adding an independently summarized callback runtime to  $L_{\text{mix}}$ .

For immediate events, the median  $L_{\text{mix}}$  is 10.87 ms in candidate Free Walk (A) and 11.34 ms in candidate Trot (B); the 99th percentiles are 25.68 and 37.58 ms. The corresponding dispatch-latency medians are 19.96 and 20.89 ms, with 99th percentiles of 34.52 and 47.40 ms. Observed maxima are 32.45 and 39.66 ms at the mix marker, but 40.87 and 48.41 ms at dispatch. Therefore, the statement that all immediate responses were below 40 ms applies only to the mix-marker endpoint in this recording, not to buffer dispatch or acoustic output. The candidate Trot period has a longer immediate-latency tail despite its more regular detection timing; this is an association between periods, not an established gait effect.

Deferred-event median  $L_{\text{mix}}$  increases from 411.57 ms in A to 443.39 ms in B. These measurements include waiting between observation and first mixing. Beat-aligned scheduling is a mechanism for such waiting in the implementation, but the logs do not isolate the cause of every deferred event. Immediate and deferred distributions are therefore kept separate; excluding deferred events from the immediate comparison does not imply that all events receive an immediate response.

**Implications for tempo adaptation.** The current source implementation accepts intervals into its BPM history only when  $0.3 < \Delta t \leq 2.0$  s. Among the retained callback-entry intervals, 69.6% in A and 93.2% in B are at most 300 ms. These figures describe how the recorded intervals compare with the nominal source-code rule, not a measured historical rejection rate: the estimator uses internal wall-clock timestamps, and neither exact BPM updates nor a contemporaneous software-version record is available in these CSVs. Nevertheless, the comparison identifies an important design limitation. An event can trigger audio while its interval contributes nothing to BPM estimation. Thus, the regular approximately 250 ms detection stream in B cannot be used as evidence that the background music accurately tracks that event rate.

**Interpretation and limitations.** Under the provisional Free Walk/Trot interpretation, the system continues detecting events and producing software audio responses in both candidate conditions. The candidate Trot period has a narrower interval distribution, a smaller immediate-response share, and a longer immediate-latency tail. These observations remain valid for periods A/B even if the inferred gait labels are later revised. They do not establish that trot caused the differences: the condition assignment is unverified, background load, speed, and possible configuration changes are insufficiently documented, and successive events within one run are correlated. No gait-specific accuracy, acoustic synchrony, real-time worst-case guarantee, or population-level statistical effect is claimed. Independent condition annotations are needed to validate the assignment; controlled repeated trials would be needed for causal or generalizable comparisons.



# 6

## Discussion

The experimental results demonstrate that the proposed framework can connect robot motion sensing with music playback in both offline and online operating modes. The offline dance mode supports choreographed, musically richer mappings by extracting motion features from recorded multi-modal data and rendering the final audio in advance. The online walking mode provides real-time footstep-driven sonification by using a lightweight sensing and audio scheduling pipeline on the robot.

The results also show that the main challenge is not average computational speed, but predictable timing. Most signal processing and trigger-detection operations are computationally lightweight. However, high-rate robotic sensing and audio playback are sensitive to missed callbacks, long-tail scheduler delays, and audio-buffer timing. This chapter discusses the implications of these findings, the role of the two operating modes, and the limitations that remain in the current implementation.

### 6.1 Platform Evaluation Scope

The real-time evaluation in this thesis used two complementary platforms. The controlled scheduler comparison was conducted on a Raspberry Pi 5, while the final deployment evaluation was conducted on the Unitree Go2’s onboard NVIDIA Jetson computer.

The Raspberry Pi 5 was used for the kernel-level comparison because the experiment required switching between a generic Linux scheduler and a `PREEMPT_RT` configuration. Installing and validating a custom real-time kernel directly on the Unitree Go2 would have been intrusive and risky, because the robot relies on vendor-provided drivers, middleware, and safety-critical locomotion software. A failed kernel modification could compromise the existing robot platform and interfere with other ongoing development work. The Raspberry Pi therefore provided a hardware-isolated environment in which the effect of `PREEMPT_RT` and `SCHED_FIFO` could be evaluated safely and repeatedly.

This does not mean that the proposed system only works on the Raspberry Pi. As reported in Section 5.4, the optimized C++ online music node was also deployed and measured on the Unitree Go2’s onboard Jetson computer using the robot’s standard operating system. The Jetson results showed that immediate footstep-to-audio events remained below 40 ms, which is well within the perceptual synchrony

target used in this work. Therefore, the Raspberry Pi results should be interpreted as a controlled study of scheduler determinism, while the Jetson results validate practical target-hardware operation.

The `PREEMPT_RT` results remain platform-dependent. The numerical improvements measured on the Raspberry Pi cannot be transferred directly to the Go2’s Jetson platform because the two systems differ in their CPUs, kernels, drivers, and background workloads. Nevertheless, the qualitative finding remains relevant: real-time scheduling reduces long-tail latency, while the decoupled C++ sensing/audio architecture enables effective operation on the robot’s default platform.

## 6.2 Motion Coverage

The current implementation evaluates a selected set of representative robot behaviors rather than the full Unitree Go2 motion library. The offline evaluation focuses on dance motions, where motor velocity, foot force, and IMU angular velocity provide clear cues for body and leg movement. The online evaluation focuses on walking, where foot-force unloading and contact patterns provide reliable gait events for real-time musical triggering.

This coverage is sufficient to validate the two central use cases of the thesis: choreographed dance sonification and online gait-synchronized music feedback. However, it does not yet cover every possible robot behavior, such as turning-in-place, stair traversal, jumping, recovery motions, or more expressive dance routines. Each additional behavior would require identifying the most informative sensor channels, defining suitable thresholds or feature detectors, and mapping the extracted events to musically meaningful audio responses.

The modular design of the framework makes such extensions feasible. New motion classes can be added by introducing additional feature extractors and mapping rules without changing the overall sensing, scheduling, and playback architecture. Future work should therefore treat motion coverage as an expansion of the mapping library rather than a redesign of the system.

## 6.3 Role and Limitations of the Offline Mode

The offline mode should not be viewed only as a workaround for real-time constraints. It is a distinct feature of the system that targets choreographed robot performances. In this mode, the robot’s motion data are recorded first, motion features are extracted from the complete time series, and the final music is rendered afterward. This allows the designer to use richer musical material, more complex arrangements, and manual refinement in a Digital Audio Workstation (DAW), while still grounding the result in measured robot motion.

This design is especially suitable for dance routines. Dance motions often contain whole-body gestures, repeated leg swings, short impacts, and moments where several limbs unload at once. These events can be difficult to sonify convincingly with only immediate real-time triggering. Offline processing makes it possible to inspect the

full motion timeline, align musical stems more carefully, and produce a coherent final soundtrack for a specific choreography.

The limitation of the offline mode is therefore not that it lacks value, but that it is less reactive. A rendered offline track is tied to a predefined movement sequence. If the robot changes timing, skips a movement, or performs a different routine, the pre-rendered soundtrack will no longer match the motion as precisely. In contrast, the online walking mode responds to live gait events and can adapt to timing variation during execution.

In the current system, the same musical source material can be reused across multiple dance modes, but each choreography still requires its own event extraction and rendering process. Future work could reduce this manual effort by automating more of the DAW marker generation, improving feature labeling, or building a larger library of reusable motion-to-stem mapping templates.

## 6.4 Real-Time Architecture and Scheduling

The real-time results indicate that the online music pipeline is lightweight enough for embedded execution, but still sensitive to scheduling tails. In the controlled experiments, median callback execution times were similar across scheduler configurations. The major difference appeared under load, where the default Completely Fair Scheduler (CFS) produced substantially larger outliers than the `PREEMPT_RT+SCHED_FIFO` configuration.

This result clarifies the role of real-time scheduling in the proposed system. A real-time kernel does not make the signal processing algorithm faster in the ordinary sense. Instead, it reduces the probability that high-priority sensing or audio callbacks are delayed behind unrelated background work. For interactive audio, this matters because a single delayed callback can cause an event to miss the next available audio buffer.

At the same time, the onboard Jetson experiment shows that a `PREEMPT_RT` kernel is not strictly required for the current prototype to function. The final C++ implementation separates the sensing path from the audio callback, keeps the footstep detector simple, and uses bounded audio-buffer mixing. This architecture was sufficient to keep immediate footstep-to-audio responses below the perceptual synchrony target on the actual robot. Thus, the practical conclusion is balanced: real-time scheduling improves determinism and is valuable for stronger guarantees, but careful software architecture can already provide acceptable performance for the online musical feedback demonstrated in this thesis.

## 6.5 Musical Quality

The proposed system focuses on synchronization between robot motion and music playback rather than on producing a professionally composed musical score. The offline mode can achieve more polished results because the extracted events can be inspected and refined before rendering. The online mode prioritizes responsiveness

and stability, using a continuously playing background track and foreground samples that are triggered or beat-aligned according to gait events.

This approach provides rhythmic consistency and avoids random sound-effect triggering, but the musical vocabulary remains limited. The current mappings use a fixed set of prepared stems and samples, and the online mode uses relatively simple harmonic and beat-alignment rules. As a result, the system demonstrates the feasibility of motion-driven sonification, but it does not fully explore composition, arrangement, timbre design, or long-term musical structure.

Future work could improve musical quality by involving musicians or sound designers in the mapping process, expanding the sample library, and evaluating the perceived relationship between robot motion and musical response with human listeners. Such work would complement the technical evaluation presented here by assessing not only whether the system responds on time, but also whether the response is musically meaningful.

## 6.6 Broader Implications for Real-Time Robotic Systems

Although this thesis focuses on audio-motion synchronization, the observed timing behavior is relevant to other interactive robotic systems. Legged robots often run many processes concurrently, including state estimation, navigation, perception, speech interaction, and behavior control. These processes compete for CPU time, and their interaction with the operating system scheduler can create long-tail delays even when the average computation time is small.

The controlled experiments show that OS-level scheduling policy can strongly affect the reliability of high-rate callback execution. Under heavy CPU load, the default CFS configuration produced substantially higher sample loss and larger worst-case callback delays than the real-time configuration. This suggests that developers of interactive robotic systems should evaluate not only mean latency, but also high-percentile and worst-case timing behavior.

The results also show the importance of architectural decoupling. By separating the high-rate sensing callback from the audio callback and by keeping the shared state small, the proposed system limits the propagation of sensing jitter into audible delays. This design principle is useful beyond music generation. Any robot subsystem that connects high-frequency sensing to user-facing feedback or control should minimize work in the time-critical callback, avoid unnecessary blocking operations, and explicitly measure timing tails under realistic background load.

For safety-critical locomotion control, stronger guarantees than those required for musical feedback would be necessary. A missed audio event may be perceptually undesirable, whereas a delayed control loop can affect stability. Therefore, the findings of this thesis should not be interpreted as a complete validation method for safety-critical control. Instead, they provide empirical evidence that real-time scheduling, careful thread separation, and latency-tail analysis are important tools for building reliable interactive robot software.

## 6.7 Ethical, Societal and Ecological Aspects

The motion-driven sonification system developed in this thesis adds musical feedback to a robot that operates in physical environments shared with people. Its evaluation therefore needs to consider more than timing performance. Relevant issues include the safe use of computing resources, the interpretation of expressive robot behavior, transparency about research methods, and the environmental costs of development and operation.

### 6.7.1 Ethical Aspects

Musical feedback must remain subordinate to the robot’s physical safety. The music node shares computing resources with other robot functions, and assigning real-time priority to audio processing does not by itself establish that locomotion or other essential tasks remain unaffected. The measured latency improvements therefore provide evidence about the sonification pipeline, rather than a safety guarantee for the complete robot. Before broader deployment, resource contention should be evaluated under realistic concurrent workloads, and operators should be able to stop music playback without interfering with essential robot functions.

The separation between the experimental and deployment platforms reflects this concern. Kernel-level experiments were conducted on the Raspberry Pi, while the onboard Jetson evaluation used the robot’s standard operating system. This avoided introducing an experimental kernel into the shared robot platform. It does not, however, remove the need to assess the behavior of the integrated system.

The technical evaluation presented in this thesis concerns robot-state recordings and software timing measurements. A future study involving listeners would introduce additional responsibilities concerning voluntary participation and the handling of recordings or identifiable information. The music tracks, samples, and external images used in the project also require clear provenance; their suitability for redistribution or public presentation should be established before such uses.

### 6.7.2 Use of Generative AI

Generative AI was used in the preparation of the thesis. Specifically, GPT-6 (OpenAI) was used to generate the conceptual illustrations of gait phases, the Butterworth frequency response, the ADSR envelope, and scheduling behavior shown in Figures 2.2, 2.3, 2.5, and 2.6. Their captions identify this assistance and distinguish the generated illustrations from the scientific literature cited as conceptual background. Generative AI also assisted with drafting and revising selected passages of the manuscript, including this section.

AI-generated explanations, code, and illustrations can contain errors even when they appear plausible. They do not constitute independent evidence for the reported results or replace the underlying literature and experimental measurements. The authors remain responsible for the final content, technical decisions, source attribution, and interpretation of the results. In particular, the AI-generated conceptual figures must not be interpreted as measured experimental outputs.

This assistance during thesis preparation is distinct from the sonification mechanism described in the thesis. The online pipeline maps detected motion events to prepared audio material through explicit rules, while the offline mode uses recorded motion and a manually arranged soundtrack. These mechanisms should be distinguished from the conversational LLM capabilities of the wider FIDO platform.

### 6.7.3 Societal Aspects

Motion-linked sound may help observers notice robot movement and could support educational demonstrations or interactive performances. However, technical synchronization alone does not establish that the robot is more understandable, enjoyable, or trustworthy. Such benefits require evaluation with human listeners. Expressive music may also encourage people to attribute emotions or intentions to the robot, although the musical responses in this system follow designed mappings rather than evidence of emotional understanding.

Public use introduces a further trade-off: sound that engages one listener may disturb another or compete with conversation and other audible information. Appropriate volume, context-dependent playback, and a convenient mute option are therefore relevant deployment requirements. Hearing differences, sensory sensitivities, and musical preferences also need consideration. Essential information about the robot should remain available through other channels, and future user evaluation should include people with different needs rather than assume that a single musical mapping is suitable for everyone.

### 6.7.4 Ecological Aspects

The environmental costs of the project include electricity for robot operation, audio playback, data processing, and software development, as well as the production and eventual disposal of computing hardware and batteries. The online system performs lightweight event detection and playback on the robot’s onboard computer. The offline mode allows a rendered soundtrack to be reused for repeated performances of the same choreography, although its preparation also requires offline processing and rendering.

These design choices describe where computation takes place; they do not demonstrate an environmental improvement. This thesis reports timing performance rather than energy consumption or life-cycle impacts, and lower latency cannot be assumed to imply lower energy use. Future evaluation could compare power consumption with and without sonification under matched robot behavior, while accounting separately for offline preparation and repeated playback.

Reusing suitable hardware and recorded sensor data during development can help limit additional equipment and unnecessary robot operation. Generative AI assistance also involves computational resources beyond the robot itself. Its environmental cost was not quantified in this work and should therefore be acknowledged without claiming a specific carbon footprint or environmental saving.

# 7

## Conclusion

This thesis presented the design, implementation, and evaluation of a real-time motion-triggered music playback system for the Unitree Go2 quadruped robot. The proposed system integrates motion sensing, ROS 2 communication, signal processing, and low-latency audio playback into a complete software framework capable of synchronizing musical responses with robot movements. To improve temporal determinism, the system was deployed under both a standard Ubuntu Linux kernel and a `PREEMPT_RT` patched kernel, enabling a systematic evaluation of kernel-level real-time scheduling for robotic audio applications.

Experimental results demonstrated that the proposed system successfully achieves reliable real-time music triggering while maintaining stable synchronization between detected motion events and audio playback. Although the `PREEMPT_RT` kernel does not significantly reduce the average end-to-end latency, it substantially decreases latency variation and worst-case response time, resulting in more deterministic execution behavior. These results indicate that operating system scheduling has a greater influence on timing consistency than on computational latency, highlighting the importance of real-time operating system support for interactive robotic applications. Furthermore, the latency breakdown analysis identified the dominant delay sources within the processing pipeline, providing valuable guidance for future system optimization.

Future work will focus on further reducing the remaining end-to-end latency by optimizing the audio playback pipeline and minimizing buffering overhead. In addition, more advanced motion detection algorithms and adaptive synchronization strategies may further improve responsiveness while maintaining robustness under different robot locomotion patterns. Finally, extending the proposed framework to support multiple robots, additional sensing modalities, and more complex interactive scenarios would further demonstrate its applicability to robotic entertainment, rehabilitation, and human-robot interaction systems.



# Bibliography

- [1] Unitree Robotics, “About go2 — developer documentation,” [https://support.unitree.com/home/en/developer/about\\_Go2](https://support.unitree.com/home/en/developer/about_Go2), 2026, accessed: Jan. 15, 2026.
- [2] E. de Visser, Y. Topoglu, S. Joshi, F. Krueger, E. Phillips, J. Gratch, C. Tossell, and H. Ayaz, “Designing man’s new best friend: Enhancing human-robot dog interaction through dog-like framing and appearance,” *Sensors*, vol. 22, p. 1287, 02 2022.
- [3] X. Chen, Z. Liao, S. Zhou, P. T. Piironen, and S. Kumar, “FIDO: An LLM-powered university campus tour guide robot dog,” 2026, under review.
- [4] B. J. Zhang and N. T. Fitter, “Nonverbal sound in human-robot interaction: A systematic review,” *J. Hum.-Robot Interact.*, vol. 12, no. 4, Dec. 2023. [Online]. Available: <https://doi.org/10.1145/3583743>
- [5] T. Hou, Y. Zhang, X. Wei, Z. Dong, J. Yi, P. Zhai, and L. Zhang, “Music-driven legged robots: Synchronized walking to rhythmic beats,” *arXiv preprint arXiv:2503.04063*, 2025.
- [6] M. Boukheddimi *et al.*, “Robot dance generation with music based trajectory optimization,” in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Kyoto, Japan, 2022, pp. 3069–3076.
- [7] C. Cuan, E. Fisher, A. Okamura, and T. Engbersen, “Music mode: Transforming robot movement into music increases likability and perceived intelligence,” *Journal of Human-Robot Interaction*, vol. 14, no. 1, p. 2, 2024.
- [8] P. R. Cook, *Real Sound Synthesis for Interactive Applications*. Natick, MA, USA: A K Peters, 2002.
- [9] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.
- [10] J. Kay, “Introduction to real-time systems,” [https://design.ros2.org/articles/realtime\\_background.html](https://design.ros2.org/articles/realtime_background.html), 2016, rOS 2 Design Documentation, accessed: Sep. 8, 2026.
- [11] —, “Proposal for implementation of real-time systems in ros 2,” [https://design.ros2.org/articles/realtime\\_proposal.html](https://design.ros2.org/articles/realtime_proposal.html), 2016, rOS 2 Design Documentation, accessed: Sep. 8, 2026.

- [12] S. A. Siewior, “Theory of operation: Real-time preemption,” <https://cdn.kernel.org/doc/html/latest/core-api/real-time/theory.html>, 2026, linux kernel documentation. Accessed: 2026-09-08.
- [13] R. Eijk, A. Kohlrausch, and S. Par, “Temporal interval discrimination thresholds depend on perceived synchrony for audio-visual stimulus pairs,” *Journal of Experimental Psychology: Human Perception and Performance*, vol. 35, pp. 1254–1263, 08 2009.
- [14] N. F. Dixon and L. Spitz, “The detection of auditory visual desynchrony,” *Perception*, vol. 9, no. 6, pp. 719–721, 1980.
- [15] R. Eg and D. M. Behne, “Perceived synchrony for realistic and dynamic audio-visual events,” *Frontiers in Psychology*, vol. 6, p. 736, 2015.
- [16] Object Management Group, “Data Distribution Service (DDS), Version 1.4,” <https://www.omg.org/spec/DDS/1.4>, 2015, accessed: 2026-09-08.
- [17] Unitree Robotics, “unitree\_ros2 low-state subscriber example,” [https://github.com/unitreerobotics/unitree\\_ros2/blob/master/example/src/src/read\\_low\\_state.cpp](https://github.com/unitreerobotics/unitree_ros2/blob/master/example/src/src/read_low_state.cpp), 2026, accessed: 2026-09-08.
- [18] —, “unitree\_sdk2 go2 lowstate dds message definition,” [https://github.com/unitreerobotics/unitree\\_sdk2/blob/main/include/unitree/idl/go2/LowState\\_.hpp](https://github.com/unitreerobotics/unitree_sdk2/blob/main/include/unitree/idl/go2/LowState_.hpp), 2026, accessed: 2026-09-08.
- [19] M. Bloesch, M. Hutter, M. A. Hoepflinger, S. Leutenegger, C. Gehring, C. D. Remy, and R. Siegwart, “State estimation for legged robots: Consistent fusion of leg kinematics and imu,” in *Robotics: Science and Systems VIII*. MIT Press, 2013, pp. 17–24.
- [20] M. Camurri, M. Ramezani, S. Nobili, and M. Fallon, “Pronto: A multi-sensor state estimator for legged robots in real-world scenarios,” *Frontiers in Robotics and AI*, vol. 7, p. 68, 2020.
- [21] T.-Y. Lin, R. Zhang, J. Yu, and M. Ghaffari, “Legged robot state estimation using invariant kalman filtering and learned contact events,” in *Proceedings of the 5th Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, vol. 164. PMLR, 2022, pp. 1057–1066. [Online]. Available: <https://proceedings.mlr.press/v164/lin22b.html>
- [22] Q. Bombléd and O. Verlinden, “Indirect foot force measurement for obstacle detection in legged locomotion,” *Mechanism and Machine Theory*, vol. 57, pp. 40–50, 2012.
- [23] A. V. Oppenheim, R. W. Schaffer, and J. R. Buck, *Discrete-Time Signal Processing*, 2nd ed. Prentice Hall, 1999.
- [24] J. G. Proakis and D. G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*, 4th ed. Pearson Prentice Hall, 2007.
- [25] S. Butterworth, “On the theory of filter amplifiers,” *Wireless Engineer*, vol. 7, pp. 536–541, 1930.
- [26] P. Okoniewski and J. Piskorowski, “Transient time reduction in time-varying digital filters via second-order section optimization,” *Applied Sciences*, vol. 15, 2025.

- 
- [27] Open Robotics, “Executors,” <https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Executors.html>, 2026, accessed: 2026-09-08.
  - [28] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, “Response-time analysis of ros 2 processing chains under reservation-based scheduling,” in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 133. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 6:1–6:23. [Online]. Available: <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2019.6>
  - [29] D. Casini, J.-J. Chen, J. Li, F. Reghenzani, and H. Teper, “A survey of real-time support, analysis, and advancements in ros 2,” 2026. [Online]. Available: <https://arxiv.org/abs/2601.10722>
  - [30] T. Hermann, A. Hunt, and J. G. Neuhoff, Eds., *The Sonification Handbook*. Berlin: Logos Verlag, 2011.
  - [31] D. Minciocchi, D. Rosenboom, R. Bravi, and E. J. Cohen, “Editorial: Sonification, perceptualizing biological information,” *Frontiers in Neuroscience*, vol. Volume 14 - 2020, 2020. [Online]. Available: <https://www.frontiersin.org/journals/neuroscience/articles/10.3389/fnins.2020.00550>
  - [32] G. Hoffman and G. Weinberg, “Interactive improvisation with a robotic marimba player,” *Autonomous Robots*, vol. 31, no. 2, pp. 133–153, 2011. [Online]. Available: <https://doi.org/10.1007/s10514-011-9237-0>
  - [33] A. Pillay, “An audio envelope generator derived from industrial process control,” arXiv preprint arXiv:2106.13966, 2021. [Online]. Available: <https://arxiv.org/abs/2106.13966>
  - [34] M. Kerrisk, “sched(7): Linux manual page,” <https://man7.org/linux/man-pages/man7/sched.7.html>, 2026, accessed: 2026-09-08.
  - [35] D. B. de Oliveira, D. Casini, R. S. de Oliveira, and T. Cucinotta, “Demystifying the real-time Linux scheduling latency,” in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 165. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 9:1–9:23. [Online]. Available: <https://doi.org/10.4230/LIPIcs.ECRTS.2020.9>
  - [36] S. A. Siewior, “How realtime kernels differ,” <https://cdn.kernel.org/doc/html/latest/core-api/real-time/differences.html>, 2026, linux kernel documentation. Accessed: 2026-09-08.
  - [37] Unitree Robotics, “Motion application,” <https://support.unitree.com/home/en/developer/Motion%20Application>, go2 developer documentation.

