



Leveraging the Web Audio API to Create an Environment Adaptive Web Based Respiratory Audio
Capture Pipeline

by
Justin Hendy

Supervisor: Nathan Morgan
Second Marker: Tim Bashford

Project submitted as part of the requirements for
the award of MSc Software Engineering (Web Development)

April 2026

Contents

1	Abstract	4
2	Introduction	4
2.1	Aims and Objectives	5
2.1.1	Aim	5
2.1.2	Objectives	5
2.2	Report	5
3	Literature Review	6
3.1	Respiratory Diseases and their Audio Biomarkers	6
3.1.1	Asthma and Chronic Obstructive Pulmonary Disease (COPD)	6
3.1.2	Asthma and COPD Audio Biomarkers	6
3.2	Remote Audio Capture for use in Clinical Settings	8
3.2.1	Shift Towards Digital and Remote Monitoring	8
3.2.2	Currently Available Systems	8
3.2.3	Issues Associated with Capturing User-Controlled Audio Recordings	9
3.2.4	The Web Audio API	10
3.2.5	Noise Reduction Techniques	11
3.3	MERN Stack	13
3.4	Why Web Audio API with DSP?	13
4	Methodology	14
4.1	Introduction	14
4.2	Research Methodology	14
4.3	Analysis and Design Activities	15
4.4	Evaluation Criteria	15
4.5	Materials and Tools	15
4.6	Procedures	16
4.7	Requirements and Design Documentation	16
5	Design and Implementation	16
5.1	Design Overview	16
5.2	System Architecture	16
5.3	Project Structure and Organisation Design	17
5.4	User Interface and Experience Design	18
5.5	Audio Processing Pipeline Design	19
5.6	Data Schema and Storage Design	22
5.7	Security and Authentication Design	23
6	Implementation	23
6.1	Audio Provider and React Context API	23
6.2	Environment Test	24
7	Results and Discussion	26
7.1	The Product	26
7.2	Audio Consistency	26
7.3	Environment Test	30

7.3.1	Static noise above maximum threshold - Fail	31
7.3.2	Static noise below the clean threshold - Pass	32
7.3.3	Static noise above the clean threshold and below the maximum threshold - Pass with filter applied	33
7.3.4	Variable noise above the clean threshold and below the maximum threshold - Fail	34
7.3.5	Variable noise below the clean threshold - Pass	35
7.3.6	Noise band-limited to the respiratory frequency band - Fail	36
7.4	Wiener Filter	37
8	Conclusions and Recommendations	38
A	Flow Diagrams	40
B	Patient User Stories	44
C	Code Snippets	46

DECLARATION

I, Justin Hendy, declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

Signed : JUSTIN HENDY

Date : 22nd April 2026

FORM OF CONSENT

I Justin Hendy, hereby consent that my Project, submitted in candidature for the MSc in Software Engineering (Web Development) degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

Signed : JUSTIN HENDY

Date 22nd April 2026

1 Abstract

The overall aim of this project is to leverage the Web Audio API in order to create a web-based respiratory audio capture system. The objectives are to design and implement an audio recording workflow, including a microphone setup guide, for use in a web browser. Apply fixed bandpass filtering to isolate respiratory audio frequencies. Implement environment testing with additional noise profile creation where necessary. Implement a Wiener filter utilising the created noise profile. Produce multiple output files representing each stage of the process and store audio files for later retrieval.

The literature review provided the basis for the project requirements and covered respiratory diseases and their audio biomarkers, remote audio capture for use in clinical settings, the MERN (MongoDB, Express, React, and Node.js) stack and the Web Audio API.

The key design decisions centred on integrating a complex audio pipeline into a user interface that guides the user through a microphone setup process. Although scoped as a proof of concept, the application includes authentication and a MongoDB schema to demonstrate production-readiness.

The application was developed following the Design Science Research Methodology, and the tool's analysis was conducted through controlled use of noise sources to induce specific behaviours, with results gathered via additional audio-recording software and log generation within the application.

The deliverable artefact is a working web-based application that meets the stated aims and objectives. This project demonstrates that the Web Audio API is a powerful tool in modern web development and that a purely web-based remote diagnostic and respiratory condition management tool is achievable.

The application was developed using the MERN stack. The application code is primarily written in TypeScript with some JavaScript where necessary.

The findings presented demonstrate that the Web Audio API is a feature-rich API that provides a great deal of tools for creating complex audio pipelines. The Wiener filter implemented provides minimal noise reduction and requires further work. The report details a number of additional features to both the audio pipeline and the application as a whole that would greatly increase the application's range of uses.

2 Introduction

Respiratory diseases and conditions continue to place a heavy burden on global populations. Data from the Global Burden of Disease 2023 study estimates that as many as 569.2 million people manage some form of Chronic Respiratory Disease (CRD), with approximately 4.2 million CRD-related deaths in the same year. In addition to personal costs, respiratory conditions impose a continuing economic burden on governments and health care providers worldwide. The National Health Service spends an estimated £800 million per year on the management of Chronic Obstructive Pulmonary Disease (COPD). In the US, estimates place COPD costs growth at \$40 billion over the next 20 years and, in 2023, respiratory disease direct costs within the European Union accounted for an estimated 6% of its total healthcare budget. The ability to continue supporting and monitoring conditions remotely is an ever-growing requirement of modern medicine, as acutely demonstrated by the 2020 COVID-19 pandemic. The need for instant personal monitoring and greater connectivity between patients and healthcare professionals has seen an increase in web-based and mobile-based health and monitoring tools, with 2019 estimates placing consumer spending on digital health products at approximately \$49 billion.

Several respiratory-based solutions on the market use proprietary hardware and/or smartphone applications such as StethoMe, iMedic, Felix, Eupnoos, and Hyfe Cough Tracker. There are a few web-based services, such as kopo.com's breathing analyser and AudibleHealth AI via coughscreening.com; both complete the audio capture process via mobile phone. These options highlight the growing use of

audio capture in respiratory and clinical contexts. Given the accessibility advantages of a web-based system, there is a need for a purely web-based service that offers, at the very minimum, web-based audio capture via desktop and/or mobile devices. The Web Audio API, available across browsers since April 2021, is a tool for real-time audio capture and processing in a web browser and provides a system suitable for developing such a product. This project will complete an engineering investigation into whether noise-responsive respiratory audio recording is attainable within the constraints of browser-based audio systems using the Web Audio API, when integrated into an adaptive, semi-automated digital signal processing pipeline with quality checking and post-processing.

2.1 Aims and Objectives

2.1.1 Aim

Design and implement a web-based audio recording system using the Web Audio API that assists users with the creation of audio recordings ready for downstream assessment.

2.1.2 Objectives

- Design an audio recording workflow with a microphone setup guide for use within a web browser
- Add fixed lowpass and highpass filters to limit bandwidth to the respiratory audio frequency range
- Implement environment testing with additional noise profile creation where necessary
- Implement a Wiener Filter utilising the created noise profile
- Produce multiple output files representing each stage of the process
- Store audio files for later retrieval

2.2 Report

Following this introduction is a literature review outlining the current burden of respiratory diseases and the need for greater access to remote detection and management systems, provide an overview of common issues encountered when recording respiratory audio along with potential solutions, compare noise suppression techniques, and discuss the Web Audio API. The methodology follows the literature review, which provides an outline of the procedures used to generate a solution for background noise detection using the Web Audio API. The following chapter outlines the project's design stage, detailing the decisions made in order to convert requirements into actionable tasks. The system implementation follows the design chapter, providing information on the tools and techniques used to develop the product, as well as details on the issues encountered. The results and discussion chapter describes the finished, working product and evaluates its performance against the given aims and objectives. Finally, a conclusion and recommendations chapter provides a summary of how the actions taken throughout this project contributed to meeting the objectives set out in the introduction, as well as a reflection on the process as a whole.

3 Literature Review

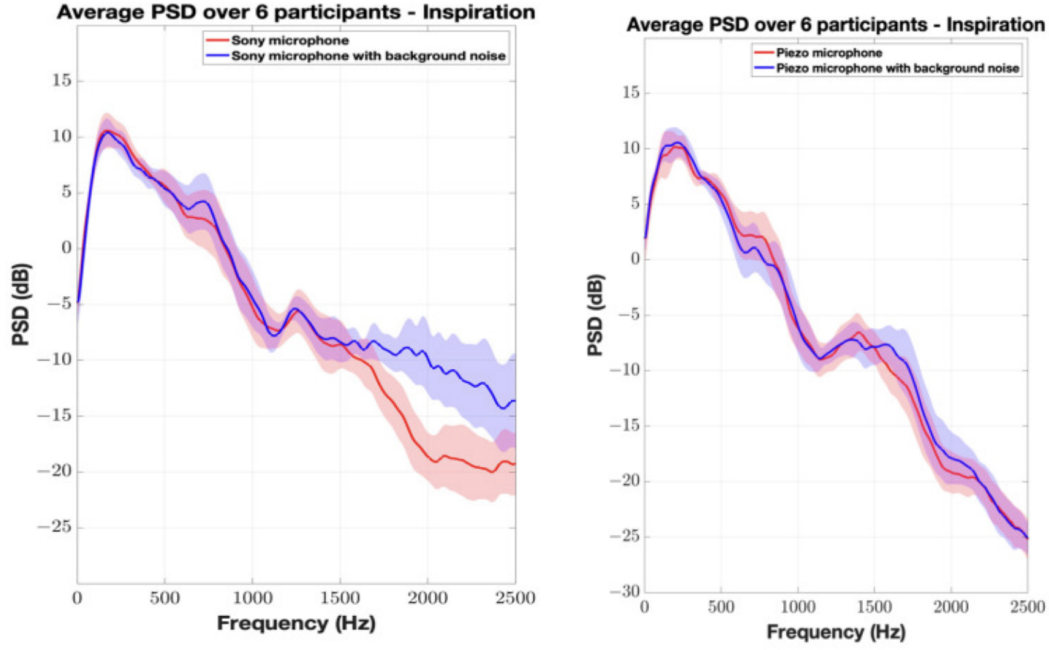
3.1 Respiratory Diseases and their Audio Biomarkers

3.1.1 Asthma and Chronic Obstructive Pulmonary Disease (COPD)

In 2017, global estimations put the number of people suffering from a chronic respiratory disease (CRD) at roughly 550 million, representing a 40% increase when compared with figures from 1990, with CRD-related deaths totalling roughly 4 million, making it the third leading cause of death worldwide [1]. More specifically, Hang et al. reported 41,555,628 new asthma cases in 2021, of which Western Europe accounted for 1.6 million. These figures represent a slight decrease between 1990 and 2021. Although asthma-related deaths during the same time period have increased, the mortality rate, how often a death occurs in a specific time period, has decreased from 9.64% to 5.2% per 100,000 [2]. Hang et al. conclude that asthma remains highly prevalent, especially in countries with higher Social-demographic Index (SDI) and among individuals under the age of 9, where SDI is a measurement of average earnings per person, schooling level and fertility rates [3]. Global figures for COPD cases vary due to differing definitions over the years, with estimates ranging from 212 to 400 million worldwide [4] [5]. Boers et al. estimate that respiratory disease direct costs in the European Union account for 6% of total annual healthcare expenditure, with 56% (€38.6 billion) allocated to COPD. Similarly, estimates relating to the United States' annual COPD costs indicate an increase to \$40 billion over the next 20 years. Within the UK, estimates place the number of people affected by COPD at around 3 million; however, as previously mentioned, COPD estimates are notoriously variable [6][7]. Rayner et al. also highlight the estimated cost to the NHS of around £800 million per year, with annual deaths of 30,000. Scully et al. sought to standardise datasets on the number of new cases of asthma, COPD and ILD (Interstitial Lung Disease) across England, Scotland and Wales, per 1,000 people per year. For 2019, England, Scotland, and Wales, respectively, saw an estimated 50,000, 3,700, and 2,000 new asthma cases; 46,000, 5,800, and 2,100 new COPD cases; and 183,000, 9,000, and 4,300 new ILD cases [8]. Both asthma and COPD continue to carry a significant health and cost burden in both the UK and global populations.

3.1.2 Asthma and COPD Audio Biomarkers

A 2024 study by Lozano-Garcia et al. sought to demonstrate the feasibility of measuring respiratory sound intensity (i.e., signal amplitude) for diagnosing and monitoring COPD, citing work by Fiz et al., who demonstrated a link between the two, particularly during inhalation [9]. Lozano-Garcia et al. used four piezoelectric contact microphones that detect sound via the movement of solid objects rather than the displacement of air particles. They placed them at various points on the subject's skin. They concluded that such measurements could detect a reduction in airflow performance among COPD patients compared with healthy patients, and that the results were consistent with peak flow measurements obtained using the traditional spirometry method [10]. Ashraf and Moussani compared the performance of a purpose-built piezoelectric microphone and a standard commercial omnidirectional condenser microphone when used to capture tracheal breathing sounds. Although they found that the piezo microphone outperformed the condenser microphone in noisy environments, the measured power spectral density (PSD), or distribution of power across frequencies, did not differ much under normal conditions, as evidenced by the two charts below [11].



(a) Condenser microphone PSD

(b) Piezoelectric microphone PSD

Figure 1: Comparison of PSD [11]

With this in mind, it is safe to assume that standard microphones can also measure respiratory sound intensity, albeit with reduced performance in noisy environments.

B.T. et al highlight that typical asthmatic symptoms include coughing, wheezing and dyspnea (difficulty breathing) [12]. To develop a system for classifying asthmatic and healthy children, B.T. et al. used recordings of cough and /:a/ (ah) sounds to train a machine learning model. Results showed that the model successfully aligned with clinical decisions, achieving 95.3% accuracy for cough sounds and 72.2% for /:a/ sounds; however, the cough model utilised 2029 samples, whereas the /:a/ model used only 333. All samples were captured using a smartphone and recorded in a noisy hospital environment. Vowel sound classification uses the identification of three formants or spaced peaks in a frequency spectrum plot and their relative intensities [13]. The chart below demonstrates average resonances of each vowel sound for men, women and children, where /:a/, as used in the word 'had', has a range of 660 Hz for the first formant for men, up to 3320 Hz for the third formant for children [14].

	Formant	heed	head	had	hod	haw'd	who'd
Men	F1	270	530	660	730	570	300
	F2	2290	1840	1720	1090	840	870
	F3	3010	2480	2410	2440	2410	2240
Women	F1	310	610	860	850	590	370
	F2	2790	2330	2050	1220	920	950
	F3	3310	2990	2850	2810	2710	2670
Children	F1	370	690	1010	1030	680	430
	F2	3200	2610	2320	1370	1060	1170
	F3	3730	3570	3320	3170	3180	3260

Figure 2: Formant Chart

Acoustic features of cough sounds include the amplitude envelope, i.e., changes in amplitude over time, and the relative intensity of grouped frequencies [15]. Ghrabli et al. studied the "unique spectral

fingerprints in cough sounds” and deemed it unnecessary to capture frequencies beyond 4000 Hz, evidenced by their use of a lowpass filter set to said frequency, and with their resulting analysis focusing on bands of 50 Hz to 900 Hz, 450 Hz to 1000 Hz and 1400 Hz to 1600 Hz of particular interest in distinguishing between pneumonia and asthma [16]. Both studies demonstrate that the spectral significance of respiratory audio lies within a specific range, from 50 Hz at the lower limit to 4000 Hz at the upper limit.

3.2 Remote Audio Capture for use in Clinical Settings

3.2.1 Shift Towards Digital and Remote Monitoring

Due to the importance of early detection and monitoring in improving quality of life and reducing hospitalisation, remote assessment via electronic devices is increasingly prevalent across various healthcare domains [17]. Tools, such as speech-based remote monitoring of asthma and COPD exacerbations, hold great promise for diagnosing and monitoring these conditions and could reduce the burden on patients and healthcare providers [17]. The ability to capture audio samples via a mobile phone has also led to the potential development of at-home spirometry tests that, when used in conjunction with AI deep learning models, can accurately correlate measured forced expiration audio to lung functionality, albeit with an error rate that is currently too high for clinical settings [18]. Alsagri et al., through their systematic literature review of emerging technologies and trends in lung volume estimation, highlight the need for more rigorous clinical testing and standardisation when assessing the impact of remote tracking devices on a patient’s health and care in order to fully realise the full potential of digital and remote monitoring [19]. Mahboubian et al. assessed the perspectives of seven key groups involved in COPD care regarding remote patient monitoring (RPM) [20]. They found that RPM offers significant opportunities to improve care, particularly in ongoing assessment between visits, resource allocation, and symptom tracking. However, health care providers noted concerns about the usability of technology and the need for clear guidelines. The key concern for patients is any change in traditional practice, such as ”reduced in-person care and overreliance on data” [20]. Yan et al. discuss the increase in the acceptance and use of smart devices in clinical settings and how the development and understanding of digital biomarkers provide opportunities to develop new diagnostic and monitoring methods. COVID-19 and the subsequent pandemic ushered in a new era of development in home-based mobile health monitoring, and the existence of telehealth and RPM will provide opportunities to improve care practices more equitably [21]. Digital and remote monitoring solutions will play a key role in future healthcare developments, and the ability to collect and correlate accurate data will be vital to safely and correctly introduce any such system into clinical use.

3.2.2 Currently Available Systems

There are existing mHealth apps for respiratory disease detection and management that focus entirely on audio capture and analysis, such as the Hyfe Cough Tracker, a smartphone-based AI cough monitoring app that can detect, record, and count coughs from cough-like sounds when used in acoustically controlled environments [22], the iMedic application, a smartphone-based self-auscultation tool for AI-Powered pediatric respiratory assessment [23] or Eupnoos, a breath analysis tool described as ”Shazam for diagnosing respiratory diseases” [24]. Several applications utilise connected devices such as StethoMe and Felix. Both StethoMe and Felix are smartphone applications that connect to a wireless stethoscope and utilise AI to provide in-home examinations. Welch et al. concluded that the Eupnoos application has ”promising” accuracy for diagnosing asthma or COPD, but cited limitations due to the small number of disease cases used [25]. Similarly, for StethoMe, scheduled clinical trials have not yet commenced at the time of writing [26]. To further highlight this point, Safavi et al. estimated in 2019 that by 2020,

consumers would spend approximately \$49 billion on digital health products; however, they found that the majority of top-funded US-based digital health companies did not measure the "clinical effectiveness" of their products or assess their use within populations considered "high-cost, high-burden" such as those suffering with chronic illnesses, older in age or diagnosed with some form of disability. Of the 156 studies, the largest group evaluated healthy patients (34); however, only 5 sought to assess clinical effectiveness. The four respiratory-related areas of asthma, COPD, ILD (Interstitial Lung Disease), and influenza garnered a total of eight studies. Of these eight, three were asthma-related, and two were COPD-related; all five studies focused on patients deemed to be high-cost/high-burden, but only two of the three asthma studies evaluated clinical effectiveness [27].

There are a few web-based services, such as kopo.com's breathing analyser and AudibleHealth AI via coughscreening.com. Kopo.com offers both smartphone and web browser capture processes; however, it only detects conditions in children aged from 6 months to 7 years. AudibleHealth AI, whilst accessible through a web browser, only offers audio capture via mobile phone.

3.2.3 Issues Associated with Capturing User-Controlled Audio Recordings

There are several issues associated with end-user or patient-captured audio recordings. In their study examining the use of cough AI systems to detect COVID-19, Khanzada et al. found that varying audio quality was the most significant barrier to developing an algorithm for this task, with particular mention of poor recording environment associated with high levels of background noise, distorted audio as a result of signal clipping (audio amplitude levels going beyond what the system is capable of handling), audio compression and wide ranging microphone quality [28]. Other issues include the need for a sufficient signal-to-noise ratio (SNR), a comparison of the wanted signal level to the unwanted signal level. When using audio recordings to determine proper inhaler use amongst patients with asthma or COPD by measuring peak inspiratory flow rate, inspiratory capacity and inhalation ramp time, Taylor et al. found that 13 dB (decibels) represented an acceptable SNR, providing 80% accuracy when estimating inhalation parameters; however, lower SNR levels, such as 10 dB, were still able to provide accuracy of around 70%. Busquet et al. discuss the importance of microphone placement, in terms of distance and angle to the user, having a significant impact on the consistency and quality of the captured audio recording [29]. Distance can affect spectral quality through the proximity effect, an increase in low-frequency spectral content caused by the lack of distance between the subject and the audio device [30], and the angle of the speaker to the device will also cause unwanted spectral changes to the recorded signal, as will the microphone's stated frequency response [29]. Below is an example polar pattern and a frequency response for a consumer condenser microphone, with the polar pattern showing changes in performance as a function of angle to the microphone transducer, and the frequency response demonstrating changes in the signal at different frequencies [31].

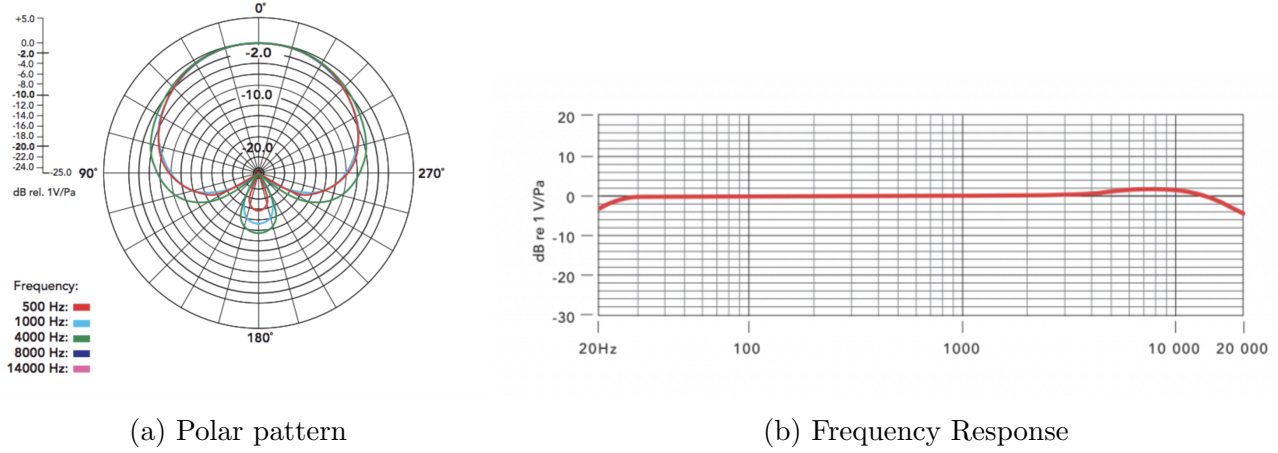


Figure 3: Rode NT1-A Polar Pattern and Frequency Response [31]

Given the current trends in remote diagnosis and management of respiratory conditions, it is clear that to fully realise the potential of user-controlled audio recordings in this setting, whether for training AI models or for human analysis, minimising the impact of these issues is vital to any such endeavour.

3.2.4 The Web Audio API

The Web Audio API, introduced in June 2021, is a W3C official web standard for audio creation and manipulation. It provides users with several "modular building blocks" that require no additional plugins, enabling the creation of a wide variety of audio-based applications for both websites and web applications on mobile and desktop devices [32]. Being a W3C standard has positive implications for long-term support and stability when creating a new platform. Further benefits include support across all major desktop and mobile browsers, without requiring users to install any additional extensions or natively installed applications, dramatically increasing accessibility and adoption, especially for users who are inexperienced or not comfortable installing software [33]. In addition to browser compatibility, Web Audio API applications can be deployed as part of Progressive Web Applications (PWAs), furthering their accessibility and use cases [34]. In 2017, Jillings et al. demonstrated the potential of the Web Audio API by developing a purely web-based digital audio workstation (DAW), using AngularJS for the user interface (UI), Web Audio for the audio engine, and a relational database for storage. A DAW is a tool for recording and manipulating multiple audio channels for playback and balancing, used across a variety of music technology disciplines such as music production and sound design. They further highlight the potential for research and analysis by collecting data on user interactions and decisions within the field of music technology [35].

The key component of the Web Audio API is the audio graph and the ability to build a modular system of nodes. The API provides access to tools for manipulating and analysing audio streams, including filtering, gain control, and real-time analysis. Examples of such nodes include `BiQuadFilterNode` for creating highpass and lowpass filters, the `GainNode`, and the `AnalyserNode`. End-to-end connectivity between inputs and outputs is available via the `MediaStreamSourceNode` and `AudioDestinationNode`, which allows for connection to speakers and/or analysis streams all within the confines of the web browser [36] [37].

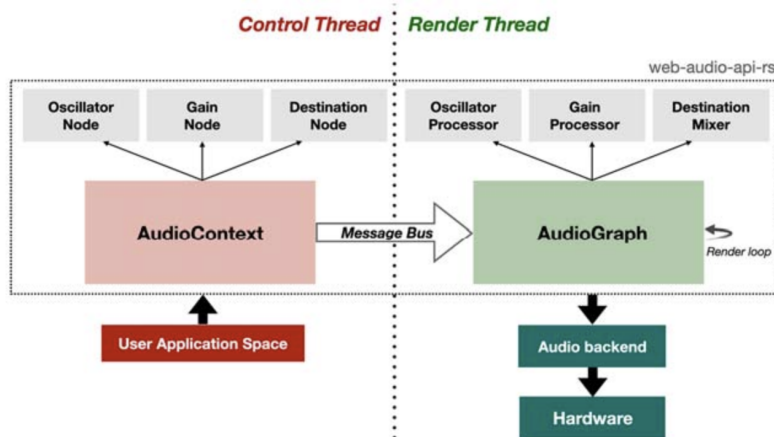


Figure 4: Web Audio API Architecture [36]

The ability to programmatically access a user’s microphone and manipulate and analyse any incoming audio stream provides developers with the opportunity to create in-browser real-time digital signal processing pipelines that can provide user feedback and support automated or user control. AudioWorklets further extend the adaptability and potential of this API by providing the ability to develop DSP processors that are not part of the standard, which run on a separate thread to the user interface (UI), providing additional low-latency (a delay between input/action and output/response) control over the audio stream [37]. The analysis node’s ability to provide live data enables the development of detailed user feedback UIs, such as waveform generation, level meters, spectrograms, etc. [33]. A W3C paper from 2013 outlines several use cases and requirements for the API standard, highlighting its potential for real-time audio manipulation, a clear indicator that the API provides opportunities to create tools for guiding and correcting user microphone usage before recording a respiratory sample for submission [38].

3.2.5 Noise Reduction Techniques

The ability to improve audio intelligibility and accuracy without adversely affecting the primary signal has many uses in speech and audio enhancement, and over the years, a large number of speech enhancement algorithms (SEAs) have emerged [39]. There are several ways to categorise SEAs, with a key definition being whether they require single-channel or multi-channel audio capture [39] [40] [41].

Adaptive noise cancelling is one such multi-channel method that, as its name suggests, can adapt to a changing noise signal and attempt to subtract the noise captured using a secondary input from a primary input containing both noise and wanted signal [42]. Given the limitations of home audio setups regarding multi-channel audio capture, this solution is not suitable within this context [43].

Basic filtering, the removal of unwanted frequencies using static filters set to specific frequencies, can provide an element of noise reduction and have several benefits [44]. As previously mentioned, they are easy to implement using the Web Audio API via the BiQuadFilterNode. In addition, they have a low computational cost due to their relatively simple arithmetic algorithm (see figure below), providing stable, predictable outputs for given inputs, which, in turn, makes them suitable for real-time audio pipelines [45] [46].

$$y_t = b_0x_t + b_1x_{t-1} + b_2x_{t-2} - a_1y_{t-1} - a_2y_{t-2}$$

Figure 5: Infinite Impulse Response (IIR) Difference Equation

While these benefits certainly have a place in any audio processing pipeline dealing with limited-bandwidth audio, such as respiratory sounds and speech, there are several limitations when specifically attenuating fixed-frequency bands in a recording. The main issue with basic filtering is the lack of knowledge about which part of the signal is noise and which is the desired signal, leading to potentially unwanted changes if applied aggressively in those key frequency bands [45] [44].

Spectral subtraction is a technique that removes noise from a signal containing the desired audio output using noise-spectrum estimates derived from non-speech audio segments. Spectral subtraction assumes that noise follows an additive model, in which the desired signal and the noise combine to produce the output signal [47]. The alternative is the multiplicative noise model, in which the noise’s strength directly alters the desired signal’s strength, or vice versa. Another assumption or requirement is that the background noise is relatively stationary in terms of spectral power and level; noise from HVAC systems, fans, steady traffic, etc. [48]. To perform the subtraction, for each input signal frequency or collection of frequencies, the noise power for that same frequency/frequencies is subtracted from the noisy input signal. Unlike standard basic filtering, spectral subtraction responds directly to the noise in a given environment. However, like basic filtering, it is computationally efficient and suitable for use in low-resource settings [47] [43]. Limitations include the introduction of “musical noise” into the resulting signal, characterised as sporadic “whistles or hums” that have a detrimental effect on the listening experience [47] [49]. Whilst its use of a static noise estimate is a benefit over basic filtering, the fact that it can only work with static noise is also a limitation, as any error in the applied noise estimate will result in an inaccurate, potentially over-suppressed desired audio signal, also known as the mean square error (MSE) [50].

Closely related to spectral subtraction is the Wiener filter [43]. Wiener filtering minimises the MSE by replacing the direct subtraction used in spectral subtraction with a weighted gain applied to the input signal at each frequency or frequency band. The weighted gain is calculated from the ratio of signal power to the sum of the signal and noise powers at that frequency (see below).

$$\text{Weight} \approx \frac{\text{Signal Power}}{\text{Signal Power} + \text{Noise Power}}$$

Figure 6: Wiener Filter Weight Calculation

With this calculation, the weighting ranges from 0 (complete attenuation) to 1 (no attenuation). When the signal power equals the noise power, the gain is set to 0.5, providing gentle attenuation of the signal. The process described here is the “optimal estimate” or “optimal statistical estimate” and reduces or minimises the mean square error (MMSE) [43]. The strengths of this approach are similar to those of spectral subtraction, but with the added benefit of less “musical noise” from weighted gain application. The limitations are also similar: it is best utilised with static or non-variable background noise and requires an accurate noise estimation/profile [51].

Machine learning, using neural networks such as convolutional neural networks (CNNs), can map degraded, noisy signals to a clean signal. Anvekar et al. compared the performance of Wiener and band-pass filtering with an autoencoder (an artificial neural network trained on clean audio with added white noise) for denoising a given audio signal contaminated with white noise. The autoencoder outperformed both Wiener and bandpass filters in terms of MSE and signal-to-noise ratio [44]. Zhu et al. demonstrate that artificial neural networks also perform well when reducing the impact of non-stationary background noise or “babble noise” [52]. This improved performance comes at a high computational cost, with Zhu et al. noting that traditional noise reduction methods, such as Wiener filtering and spectral subtraction,

are still widely used due to their low computational cost and effectiveness in pre-processing environments [53] [52]. ML models often require sensitive data for training, and concerns exist about the safety and security of that data, as machine learning is vulnerable to model inversion and extraction attacks, as well as membership-inversion attacks [54].

Given the relative strengths and limitations of each technique, Wiener filtering would suit a real-time, lightweight, browser-based audio enhancement pipeline. To increase the effectiveness of such a system, adding basic lowpass and highpass filtering with conservative cutoff frequencies to the incoming signal will remove much of the unnecessary bandwidth from the frequency spectrum, thereby reducing the attenuation required by the Wiener filter. In addition to such measures, a noise power and variability check can ensure that users recording in uncontrollable environments beyond the Wiener filter’s operating window are not permitted to continue, preventing the production of an audio file unsuitable for further processing and assessment. To determine signal strength and variability, time-based signal analysis will be required.

A standard approach to audio signal analysis involves creating short, fixed-length frames containing collections of audio samples. Any data derived from that frame then represents the entire frame, as any change is assumed insignificant over such a short period. Measurements taken from each frame are then compared across larger collections of frames to assess variability within a signal. In audio signal processing, frame lengths of 20-40ms balance time resolution with local stationarity assumptions, with 25ms being a commonly adopted choice [55] [56].

3.3 MERN Stack

MERN is a technology stack for developing web applications and includes MongoDB (database), ExpressJS (backend framework), React (frontend library) and NodeJS (server environment). MERN applications are deployable on mobile devices and web browsers and known for their responsiveness, thanks in part to React’s Virtual DOM (Document Object Model - programming interface for representing a web page) and Node’s non-blocking I/O (prevents input and output tasks like reading a file from blocking further code execution) [57]. Other stacks are available, such as the MEAN stack (replace React with Angular), and whilst there are differences (e.g., the use of the virtual or real DOM), the key reason for choosing one over the other is author familiarity. When working with audio files of unknown size, MongoDB’s GridFS system supports storing files larger than 16 MB. GridFS also optimises retrieval by splitting files into chunks, distributing them, and reconstructing them as needed [58]. NodeJS is a JavaScript runtime environment that allows JavaScript code to execute outside the browser, thereby enabling app development via a unified language [59]. ExpressJS is a server-side framework for NodeJS and provides methods for handling different “HTTP requests, and parsing URL parameters”, and provides no more than the core functionality required to implement HTTP-based APIs [60]. ExpressJS is an extremely popular framework, as evidenced by its current weekly downloads exceeding 58.5 million (at the time of writing). Other frameworks, such as NextJS, NestJS, and KoaJS, each have benefits of their own, such as NextJS’s folder-based page routing structure; however, as with React vs Angular, author familiarity is key.

3.4 Why Web Audio API with DSP?

Given the previously outlined issues with user-controlled respiratory audio capture, conventional DSP implemented via the Web Audio API offers a good opportunity to deliver a multi-platform, multi-device solution for real-time audio quality enhancement.

Respiratory audio biomarkers are subtle and sensitive to poor recording environments, such as background noise, low signal-to-noise ratios, clipping/distortion, and inconsistent positioning and technique.

However, the biomarkers also exist within known frequency ranges and, given the right conditions, can be captured successfully using consumer-based technology. Whilst post-processing solutions, and in particular ML models, offer opportunities for enhancement that surpass the capabilities of conventional DSP, they require substantial computational power, are privacy-sensitive, and cannot provide the user with real-time feedback. The goal at the start of a respiratory audio tool is to ensure the captured audio is of the highest quality. From here, ML models and other analysis tools can extract features and markers from the audio where required; however, downstream performance or output will diminish if the audio is not of a particular standard.

The Web Audio API provides real-time audio graph processing with low-latency audio nodes for standard audio DSP control. Beyond this, access to both frequency and time-domain data enables developers to build real-time user feedback UIs and data-controlled pipelines. The `AudioWorkletProcessor` class enables developers to build bespoke DSP units, such as a Wiener filter. The modular nature of the Web Audio API also allows for a clear understanding of each stage of the pipeline and prevents the "black boxing" of specific processes. It allows for deterministic behaviour: the system will produce the same output for a given input, and this will apply across every process/step of the pipeline. All JavaScript/AudioWorklet code executes locally in the browser runtime; all audio samples from the user's device microphone are passed through the audio graph and processed in real time. Hence, avoiding machine learning at this stage ensures all processing is on-device, improving privacy and reducing data transmission until the file is ready for storage. Any submission of files will require the user to take direct action. Beyond the technical capability, the API is a cross-platform standard, requires no installation and offers immediate access via consumer devices.

4 Methodology

4.1 Introduction

The purpose of this study is to investigate the use of digital signal processing techniques in a web browser audio pipeline that utilises the Web Audio API, in the context of building an end-user respiratory audio capture system. The developed artefact will consist of an environment noise level and variability testing system, a signal level check with automatic gain adjustment, audio capture capability, automatic audio file quality checks, audio post-processing, and storage of various audio files captured at each stage of the process.

4.2 Research Methodology

The chosen methodologies for this study are the Design Science Research Methodology (DSRM) alongside experimental testing. This study aligns with DSRM, where the structured process of problem identification, objective definition, design and development, demonstration, evaluation, and communication align with the stated aims and outcomes of this study [61] [62] [63]. Producing a research contribution is a significant aspect of the DSRM, with the overall aim being an artefact and solution to an identified and "understood research problem", and where the artefact can consist of any number of outputs, such as a new method or technical product [63].

Experimental testing allows independent variables to be tested in controlled environments to ascertain their effect on dependent variables [64]. In the context of this study, the experimental tests need to demonstrate that, under different environmental conditions and user signal levels, the developed application responds appropriately. The experimental tests also need to demonstrate that the defined audio defects are correctly identified post-recording and dealt with accordingly.

4.3 Analysis and Design Activities

The literature review in Chapter 2 identified problems caused by poor-quality remote respiratory audio capture, whether for clinical assessment or for passing to an ML model. Issues included poor microphone placement, suboptimal recording environments due to background noise, signal clipping, and poor recording levels. The literature review also identified the lack of web-based respiratory audio services, with many current respiratory-based remote monitoring and diagnosis applications requiring downloading and installing on a mobile device, as well as some form of proprietary hardware. These issues defined the audio pipeline’s functional requirements, which in turn became user stories.

The client-server system architecture supported browser-based audio capture and storage via the Web Audio API. Prior to implementation, an end-to-end capture and processing workflow outlined each stage of the pipeline, from microphone to storage and retrieval. Where appropriate, literature-derived criteria and values defined the decision logic for the environment test, but further refinement occurred through iterative testing throughout the implementation. Ensuring visual feedback on the live audio signal level was key to the user interface design, as was providing clear instructions for each stage of the environment test.

4.4 Evaluation Criteria

There are four stages in the pipeline to determine whether the user can capture audio and whether any processing is required to improve its quality: environment test and applied noise profile, signal-to-noise ratio test, captured audio quality tests, and applied post-processing. This study will focus on environmental testing and the subsequent application and performance of a noise filter when required. The environment test performs two functions: it not only assesses background noise to determine whether the user’s environment is suitable for respiratory audio capture, but also captures a background noise profile for future use if the noise falls within certain criteria. Depending on whether the environment meets usability thresholds, the system will either pass, pass but with a noise profile captured, or fail the environment under test. The testing method involved simulating various background noise environments, with each simulation chosen for specific characteristics that allow testing against specific criteria. Tests will also assess behaviour close to threshold levels to ensure correct categorisation. Each test generated several system logs and, where necessary, visual spectrum plots showing relationships among noise levels, noise variability, noise frequency distributions, and evaluation outcomes, supporting verification of system robustness. The second stage of the pipeline consists of a signal-to-noise ratio (SNR) test and subsequent automatic gain control. The evaluation criteria consist of measuring the input signal level and either rejecting the signal or applying gain, as needed. If the signal requires too much gain to reach the minimum target level or is too loud, the test fails. If the signal requires a small amount of gain to reach an acceptable, usable level, gain is applied. Finally, the signal level is compared with the median noise level from the first test to determine whether the signal-to-noise ratio is sufficient. Upon completion of a recording, several evaluation criteria will assess the pipeline’s performance. Noise reduction evaluation requires the use of difference spectrograms generated in Audacity (a free, open-source audio recording and editing software package), where the difference between two versions (pre- and post-noise reduction) of the same recording visually demonstrates any reduction that has taken place.

4.5 Materials and Tools

This study used a MERN (MongoDB, Express, React, and Node) stack with TypeScript, providing a type-safe framework for rapid web-based prototyping to develop a web-based environment noise test.

React, Vite, Tailwind CSS with Shaden, and React-Router implemented the user interface and managed page navigation. The Web Audio API provided the necessary tools to develop the DSP pipeline for environmental noise testing and analysis through its modular routing and audio node system. Use of the Winston logging library enabled the development of JSON logs that provided structured test results to support the verification and analysis of outcomes. Microsoft Excel enabled the conversion of JSON noise profile data points to visual frequency plots.

4.6 Procedures

As the requirements were primarily defined upfront based on the literature review, the development process loosely followed a Waterfall approach [65]. However, iterations refined the decision-logic variables to ensure the system correctly categorised environmental noise, signal level, SNR and audio quality, aligning with Rapid Application Development (RAD). RAD consists of iterative refinement, evaluation, and artefact development [66]. Structured approaches such as Yourdon or Ward-Mellor emphasise the need for data flow diagrams and formal modelling, but are unnecessary for developing a single pipeline [67]. The Agile/Scrum approach requires stakeholder input, sprints and reviews. Given that this is a single-developer project with no stakeholders, this again does not fit the purpose of the study [65].

4.7 Requirements and Design Documentation

Requirements derived from the literature review provided the basis for a series of user stories acting as lightweight specification and design documentation. Further design documentation includes an environment test logic and decision flowchart (see appendix), as well as a high-level user interaction and noise capture lifecycle state machine model.

5 Design and Implementation

5.1 Design Overview

The overall design requirement is to develop a browser-based audio-capture platform specifically for respiratory audio analysis. To ensure user-captured audio is suitable, an environmental noise assessment (a process for measuring unwanted background sounds) will be conducted before each attempted recording. Based on the measured noise levels, recording may be blocked, allowed, or allowed with the application of a Wiener filter (an algorithm that estimates and subtracts unwanted background noise). After the environment test, users will receive guidance on adjusting their audio input volume. This helps achieve a good signal-to-noise ratio (a measure comparing the level of the desired audio signal to the level of the unwanted noise) and helps to prevent artefacts, such as clipping (which occurs when the audio signal is too loud and exceeds the maximum system level) or distortion (where the audio is changed from its original form), in the submitted file. A key design aspect focuses on providing a simple interface for recording once all tests pass. After recording, the system must perform a quality check before allowing the audio to be submitted for analysis or storage. To maintain transparency, it is also important to allow audio to be captured and stored at various pipeline stages (meaning at each step in the audio processing workflow). Finally, the application requires security, user authentication and authorisation.

5.2 System Architecture

The system architecture is a browser-based audio capture system which employs a separated client-server architecture with a real-time audio processing pipeline, RESTful APIs, document storage, and external

analysis integration.

The client-server separation has several benefits, such as independent technology for frontend and backend services, enabling flexibility in upgrading or replacing specific components. The server can also handle multiple client types simultaneously through exposing HTTP endpoints that any client capable of making HTTP requests can consume, such as a React web app, a mobile app, a research dashboard, or a command-line interface tool. There are several trade-offs with this architectural pattern, including the need to maintain separate codebases, configure cross-origin resource sharing (CORS), and use separate build processes and dependencies. The client-server architecture makes API versioning critical when adding or upgrading endpoints, as different clients may interact with the service via different versions of an endpoint.

Another key tenet of the system architecture is real-time audio processing with external analysis integration. The ability to provide real-time audio processing via the Web Audio API is vital, as it enables both user- and app-controlled adjustments to the audio signal chain in response to the immediate environmental conditions and the captured respiratory audio signal level. The external analysis service, VoxLab, is integrated as a demonstration of potential downstream services that would utilise an audio capture pipeline such as this.

As previously mentioned, MongoDB, ExpressJS, React, and NodeJS form the MERN stack, which is the chosen full-stack development and build environment for this project. The flexible schema approach in MongoDB, through its binary JSON (BSON) document model, enables rapid prototyping and is particularly useful in proof-of-concept scenarios where complexity and validation can be added as the application grows. Beyond the benefit of schema flexibility, MongoDB, in combination with its GridFS specification, is extremely effective for storing audio files, which, given the focus of this study, elevates MongoDB over other database options. The JavaScript ecosystem provides access to numerous libraries and frameworks that simplify both frontend and backend development. The NodeJS runtime environment enables JavaScript throughout the codebase and, when combined with the ExpressJS framework, provides developers with a number of features for server-side web application development, including flexible routing, middleware, and support for RESTful API design. The ReactJS JavaScript library is a tool for developing dynamic and interactive user interface components where reusability, performance and maintainability are central to the development process. TypeScript is a superset of JavaScript and can be used with the MERN development stack. The key benefit of using TypeScript is the addition of static typing, which allows errors to be caught earlier in the development process and ultimately leads to a more maintainable codebase.

5.3 Project Structure and Organisation Design

The three key elements of the client folder structure will be the features, pages, and app folders. The client structure is feature-based because there is limited overlap between the two main application features: authentication and audio recorder. The audio-recorder feature will manage hooks, utilities, types and components relating to Web Audio API integration, real-time processing, audio-visual feedback, and the logic for environment and signal-to-noise ratio testing, whereas authentication will manage JWT token handling, login/registration components, protected route logic, and session management. A layer-based approach, where files are grouped by type (e.g., all hooks together and components together), would become unwieldy and hard to maintain as features grow in complexity. This feature-driven approach also provides clear development boundaries and creates a structured way to extend the application with new capabilities. Certain hooks and components will not fall under the scope of any single feature and will be shared across the application; therefore, they will be placed in their own folders alongside the features, pages and app folders.

The pages folder will hold top-level route components and page layouts. Again, this structure

provides a clear development strategy: any new page layouts must be placed in their own new folder within the pages folder, or nested within an existing folder if accessed via another page. Page layouts will primarily return JSX alongside feature- or generic UI components, and in some cases, React Router navigation elements.

The app folder contains the application scaffolding or infrastructure that supports feature development. This includes centralised state management via providers, consistent page layouts with reusable headers and content areas, the routing structure, and the root App component that orchestrates the overall application structure.

The React Router library is a solid option for building a routing structure and supports multiple pages within a single-page application, handles URL parameters and query strings, provides tools for creating protected routes that require authentication, and supports nested routes and layouts. Alternative solutions include TanStack Router or Next.js router; however, using the Vite build tool with its fast development server and optimised hot module replacement precludes Next.js as a routing option, as Next.js includes its own integrated build system and while TanStack router has many benefits, it is also more complex and comes with a steep learning curve when compared to React Router.

Employing centralised state management will enable multiple features and components to share context/state, ensuring a single source of truth where necessary. Centralised state also minimises prop drilling - the practice of passing data through multiple component layers as props, even when intermediate components don't need that data.

Unlike the client, the server will utilise a standard layered structure with four key layers: Routes, Controllers, Models, and Utils. Routes will contain files defining API endpoints for the three main application actions: audio storage and retrieval, audio analysis, and user actions (login, logout, etc.). Controllers will contain files that define the business logic for handling each client API request and, where necessary, returning a response. The models folder will contain all data schema definitions and data validation. The utils folder will contain shared functionality to reduce code duplication. This approach is suitable because of the limited number of features, making it easier to find all controllers, models, and routes in one place while maintaining clear separation of concerns.

5.4 User Interface and Experience Design

The user journey will start with either registration for first-time users or logging in with a username and password. Once authenticated, the user can initiate a microphone test, complete a recording and submit the results for storage and analysis. To aid user navigation, components such as breadcrumbs and a header navigation bar will be present across all pages. As the primary purpose of this app is to assist audio recording, the user interface will need controls for selecting a microphone source, starting the microphone within the application and monitoring the microphone signal level via an audio level meter. The audio level meter will require coloured segments indicating signal strength: green for low to moderate, yellow for optimal to approaching clipping, and red for clipping. Providing numerical data on signal strength is not vital from a user experience, but it will provide useful feedback during development and testing. To carry out the necessary tests, the user will need to follow instructions, so a clear panel for text that updates based on test states and results is vital. Upon completion of an attempted recording, a panel displaying the results of the quality checks will provide useful insights, particularly if the recording failed for a specific reason. Finally, buttons for starting and stopping recordings, clearing the audio for when the user is not happy with the result and a space for labelling and uploading the completed audio recording.

5.5 Audio Processing Pipeline Design

The audio processing pipeline is a key feature of this application and therefore requires additional attention. The signal will need to flow from the user's microphone through several processes, then be recorded and stored for future retrieval and analysis. Once the user has selected their microphone, a number of constraints must be applied to set it up, such as channel count, sample size, and sample rate, as well as constraints that prevent system-wide behaviours from altering the audio signal, such as echo cancellation, noise suppression, and auto gain control. All of the aforementioned behaviours, except channel count, sample size, and sample rate, can alter the audio signal and must therefore be disabled to provide audio that has not been changed outside of the application's control, as there will be no way to ascertain what has been changed as a result of these actions.

Channel count defines the number of audio channels, which, given the purpose of this application, will always be 1 (mono). It should be noted that, according to MDN, this constraint is sometimes ignored by certain browsers; therefore, the audio stream must be manually downmixed to mono within the application to ensure consistency [68]. The sample rate determines how often the incoming audio signal is sampled per second. An increase in the sample rate results in more samples and, therefore, a larger file size. Sample rate is measured in Hertz (Hz), and typical values range from 44.1 kHz (CD-quality) to 192 kHz. Sample size or bit depth determines the vertical resolution of the sampled audio signal; it is expressed in bits, such as 16, 24, or 32. These values represent the number of binary digits available to store each sampled value with 16-bit resolution, giving you 2^{16} or 65,536 different values, with each value taking 16 bits of storage or approximately 2 decimal Bytes. Higher bit values also improve a system's dynamic range (the distance between the noise floor and the upper limit). Given the trade-off between higher-quality audio and the relatively short recording lengths for respiratory analysis, 24-bit 48 kHz will produce reasonably sized, high-definition audio files. For example, a one-minute mono recording captured at 24-bit, 48 kHz resolution will produce a file size of approximately 8.64 MB (decimal MegaBytes) or 8.24 MiB (binary MebiBytes).

The Web Audio API provides a number of basic audio operations via audio nodes that can be linked together to form an audio routing graph through which an audio stream can be processed or analysed. Basic operations include gain control, filtering, reverb, compression and panning. In addition to these basic audio operations, the Web Audio API provides the facilities for creating bespoke, more advanced processes, such as downmixing a stereo signal to mono or creating a signal-adaptive noise filter. Below is a flowchart that shows the path of the audio signal from the microphone through the audio routing graph to a number of destinations that allow recordings to be made at various points along the graph. The basic operations required are gain control and filtering. Gain is straightforward: it enables the application to adjust the signal level. Filtering will perform two functions: DC offset and band limiting. DC (Direct Current) Offset filtering protects against inaudible electrical voltages that can occur during analogue-to-digital conversion and prevents the incoming audio signal from being distorted by being offset from zero. This can cause issues with peak detection, as one side of the waveform will appear to have a higher amplitude than the other. As DC Offset is a constant shift that never changes, it has a frequency of 0 Hz and can be removed with a high-pass filter set very low, below the range of human hearing, thereby minimising any impact on audio quality. Band limiting the signal will allow for the rejection or reduction of non-respiratory significant frequencies. Respiratory-significant frequencies tend to reside within the 50 Hz to 2500 Hz range; therefore, limiting the audio signal to a range slightly wider than this will ensure non-respiratory sounds will be reduced or removed in the final submitted audio. Before looking more closely at the audio pipeline, it's important to introduce some standard audio system terms for signal measurement.

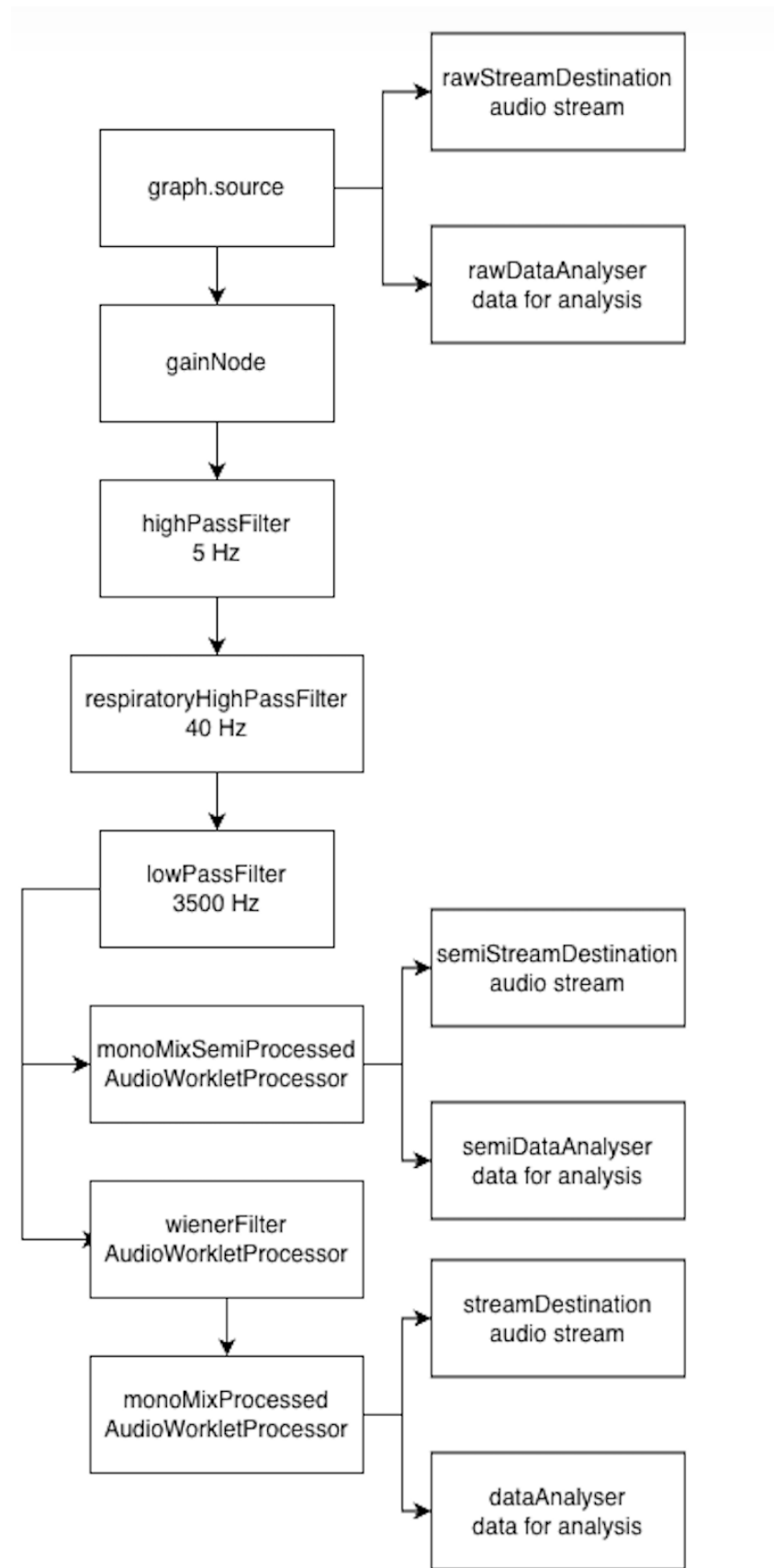


Figure 7: Flowchart - Audio Graph

The terms are amplitude, gain, peak level, root-mean-square (RMS), decibel (dB) and dBFS (decibel

level full scale). Amplitude is simply the measurement of a signal’s strength, and gain is an amplification factor or a measurement of a change in signal strength. The signal peak level is the maximum instantaneous level of a signal over a defined period. RMS is a signal’s average power and more closely represents how we perceive the loudness of a signal over a given time interval. RMS is calculated by squaring each sample value, averaging the squared values, and taking the square root of the result. The decibel is a relative unit of measurement that represents the ratio of two signals and, therefore, can be applied to different systems using different reference points; all of the previous signal measurements (amplitude, peak and RMS) can be given in decibels. Full-scale decibel measurements are used in digital systems, where 0 dBFS is the absolute limit of the system and the point at which signal clipping occurs. 0 dBFS is the reference point for this decibel ratio measurement, and therefore, all usable levels are below 0 dBFS. Other decibel systems include dB SPL (used for sound pressure measurements), dBu (voltage measurement used in audio), and dBA (a weighted measurement that accounts for how humans perceive loudness at a given frequency).

The main purpose of the audio processing pipeline is to provide noise suppression and gain control over an incoming audio signal. Enabling greater control over the design and implementation of these processes is a key aspect of this design. Noise suppression in this pipeline involves an environmental test followed by a Wiener filter to target and reduce unwanted noise from the signal. There are several node packages that provide noise suppression or cancellation, as well as just applying the MediaDevice constraint; however, to provide the greatest flexibility and fully utilise the Web Audio API, a bespoke Wiener filter will be developed and added to the audio routing graph. The Web Audio API AudioWorklet interface allows for bespoke audio processes to run on threads separate from the main thread by passing messages between each thread, and provides very low-latency real-time audio processing [68]. Once the process script is in place, integration with an existing audio graph is as simple as adding pre-existing audio processors such as the Biquad filter. This flexible interface will allow for the addition of both a mono downmix processor and a Wiener processor to the audio graph.

With a Wiener filter script in place, an environment test will determine whether to engage the filter and provide a means to capture a noise profile from the signal. There are three checks that must be performed during this environment test. Is the noise too loud, is the noise too variable, and, finally, if the noise is significant but not too loud, is the noise dominated by non-respiratory frequencies? In order to carry out each check, the user will need to start the test and not make any sound so that the system can “capture” a snapshot of the current environment. From here, the application will determine the RMS of the environmental noise and compare it with a given maximum noise level; if the noise exceeds the limit, the application will return a fail. To check the signal’s variability, a series of RMS values taken at equally spaced intervals, arranged in ascending order, will allow the application to determine the difference between the 90th- and 10th-percentile RMS values. If the difference is too great, then the signal will not pass the variability test. The final check will ascertain whether the signal is dominated by frequencies above or below the respiratory band and whether it is loud enough to be considered relevant. If so, then the signal will pass, but a filter will need to be applied. At this point, the noise spectrum’s frequency response needs to be stored and passed to the Wiener filter.

Audio gain control helps ensure the recorded signal is at an optimal level, i.e., with a good signal-to-noise ratio, no distortion or clipping, and a sufficient level for listening and analysis. In a clinical setting where repeatability, transparency, control, and predictability are vital, using the browser audio constraint `autoGainControl` is not suitable. Auto gain control (AGC) is essentially a black box, with the aim of maintaining audio level consistency, offering no insight into the thresholds, no system for capturing the original audio signal, or whether it will behave the same across different systems or during repeated recordings of the same test. As previously mentioned, audio level consistency is not the aim in this context; rather, the aim is to prevent signal clipping and a poor signal-to-noise ratio. To assist the user with such a setup, the test will need to perform three signal level checks and allow the user to

make adjustments.

The first signal-level check ensures there is sufficient headroom, or margin, between a system's normal operating level and the maximum level it can handle before clipping occurs. The application will need to monitor the peak level of the incoming signal and warn the user if it exceeds a certain threshold. Anything below 0 dBFS is sufficient, but it is important to utilise the full dynamic range of 24-bit audio recording; therefore, there is no need to allow signals close to 0 dBFS. It is also vital to be mindful of sudden spikes in level that will exceed the detected maximum peak level. Once the signal is deemed low enough not to cause digital audio clipping, it will need a sufficient level to be useful; therefore, peak level is not an appropriate check in this scenario, as it doesn't provide a useful measurement of perceived loudness. The signal's RMS value will provide this information and must exceed a specified lower limit. The final check will determine whether the audio signal has a sufficient signal-to-noise ratio by calculating the difference between the RMS values of the wanted and unwanted signals. Once calculated, it must meet a predetermined threshold. Another important aspect to consider in relation to SNR is that a single value representing the RMS of the entire signal provides no information about whether the signal and noise overlap in the frequency domain and, given the use of a Wiener filter to help reduce the spectral significance of any static noise, an option to pass a signal if the respiratory frequencies dominate non-respiratory frequencies is extremely useful within this context. All three measurements taken from the audio signal will allow the application to either pass the audio setup or ask the user to make adjustments based on the outcomes. Once the test is complete, there are only 2 possible adjustments: adjusting the microphone level using the system controls or physically moving closer to the microphone. If the signal is too quiet or too loud and the signal-to-noise ratio is not an issue, adjusting the microphone level will solve the problem; if the signal-to-noise ratio needs improvement, moving closer to the microphone relative to the source of the noise is the only solution.

The final stage of the pipeline will provide the user with a single upload point that stores audio files from each processing point in the pipeline in the connected MongoDB and submits only the fully processed audio file to a third-party vocal analysis tool called VoxLab.

5.6 Data Schema and Storage Design

The database technology used for schema definition and validation is MongoDB with Mongoose ODM (Object Data Modelling), alongside GridFS for large-file storage. Mongoose ODM is a Node.js package that translates between NoSQL document structures and application objects, similar to how ORMs (Object-Relational Mapping), such as Hibernate or JPA, provide for SQL databases. Mongoose provides comprehensive SchemaTypes and validation options that allow developers to set a range of parameters for each field, such as enum values, custom text field validation, minimum and maximum lengths, required fields, and automatic trimming. Each option can also return customisable error messages when validation fails. Middleware that can execute before or after specific operations, such as save or find, as well as custom instance and static methods, can be created to encapsulate model-specific business logic and data manipulation functionality.

The User model will demonstrate these features through its schema design. Data fields will include firstName, surname, email, and role. The role field will use enum values (admin, patient and clinician) to enable clear role-based access control. To improve search functionality, lowercase versions of names will be generated automatically via pre-save middleware. Password security will be handled using bcrypt hashing, with additional fields for the password's last change date, a password reset token, and a password reset expiry date. The model will use an active boolean field to implement soft deletion, with query middleware automatically filtering out inactive users from all find operations. Custom methods will provide secure password comparison, token validation, and reset token generation functionality.

The primary purpose of the Audio model will be to link the four MongoDB object IDs for the

recordings captured at various stages of the audio pipeline into a single record. Additional fields will store a user identifier, a filename, and an upload date. To store the actual audio binary data, GridFS creates two collections: one for the audio file metadata, suffixed with `.files`, and another for the individual binary chunks, suffixed with `.chunks`. GridFS stores audio by splitting large files into 255 KiB (261 KB) chunks. The chunk documents also include a field for the GridFS-generated audio file ID. The GridFS files collection automatically stores data on the full size of the audio file, the chunk size (which is fixed by default and the same for all documents), and the upload date. When creating the GridFS audio stream, you can pass additional information, such as a filename and any additional metadata. With these options, it would be possible to omit an additional Mongoose Audio model; however, given the need for four audio streams per recording, it is important to have a document that links them. As previously mentioned, a 24 Bit 48 kHz one-minute audio recording will require roughly 8 MB of storage, which is only half the limit before GridFS becomes necessary. The benefits of utilising GridFS in this scenario are that it future-proofs the service, allows for storage of larger files both in terms of length and quality, has built-in metadata, and has better memory efficiency when streaming audio. The final benefit is currently not utilised within the application; however, future features will involve clinicians listening to captured audio, and improved streaming will prevent the need to download audio to a local machine.

5.7 Security and Authentication Design

Security and authentication will utilise JSON Web Tokens (JWTs), with both an access and a refresh token generated when a user logs in via the login page or registers as a new user. Access tokens will be stored in authorisation headers as Bearer tokens, will have a short lifespan of 15 minutes, and will allow certain routes to be protected based on the permissions associated with specific roles. Refresh tokens will be stored in HTTP-only cookies and have a 1-day lifespan. Refresh tokens allow the application to generate new access tokens when an access token expires after 15 minutes or on page refreshes. To increase the refresh token's lifespan, a user can select "Remember me" in the UI, which sets the refresh token's age to 30 days.

6 Implementation

6.1 Audio Provider and React Context API

There are several areas of the application that ultimately control and define how it assists the user in preparing their audio system for capturing audio and submitting the audio for further analysis. Central to controlling state and user interactions is the AudioProvider (see code appendix). The AudioProvider utilises the React Context API for centralised state management, allowing state to be shared throughout the codebase without passing props directly between components and functions. During the early stages of development, individual components and hooks used the React `useState` hook to store and retrieve data, and when combined with prop drilling, the state was passed from a parent to a child component as needed. Once the audio pipeline grew beyond a simple audio recorder, it became apparent that managing state through components and prop drilling would not suffice and that certain data needed to be held centrally. Initially, all data related to the audio pipeline was moved to the AudioProvider, which became the single source of truth. If a component, such as `RecordingControls`, or a bespoke hook, such as `useMicSelector`, required access to various audio-related data, it could retrieve it by calling `useAudioState`, and subsequently, any state held within the AudioProvider would be available to that component or hook. This not only simplified the interfaces for each component but also ensured the

correct data was available. In total, 15 components/hooks utilised the `AudioProvider`. This alone made the `AudioProvider` essential to the application; however, once environment noise testing was introduced, it provided the perfect location to control the flow of the environment test and subsequent communication with the Wiener filter when a noise filter is required.

To perform an environment test, the application needs to coordinate among the user interface, the microphone audio stream, and the Wiener filter. Once the user selects the microphone (if multiple options are available) and starts it, the `AudioMeter` component visually updates to indicate that a signal is present via a level meter, and a test prompt appears, replacing the mic setup controls. There are two states managed by the `AudioProvider`, `MicState` and `TestPhase`, that control the various components involved in this process, ensuring that the correct text appears and that the functions associated with a particular button press result in the correct action. The combination of `MicState` and `TestPhase`, in conjunction with the `AudioProvider`, forms a finite-state machine that controls the entire flow. The `MicState` has six transitions: `idle`, `micEnabled`, `testPrompt`, `testing`, `ready` and `error`. `TestPhase` provides several stages for both the environment test and the signal-to-noise ratio test; however, the signal-to-noise ratio test has not been implemented. The `TestPhase` not only controls the behaviour of the application but also combines with `testPhaseInstructions`, which holds all the text to display with each phase. Once the user initiates the environment test, the application invokes the `environmentTesting` utility, which contains all the logic for capturing a sample of the environment audio, analysing it, and returning one of three results to the `AudioProvider`. The three results are `pass`, `pass filtered`, or `fail`. From here, the `AudioProvider` can coordinate with the UI to again update the text displayed to the user and change the actions associated with the available buttons. For example, if the test fails, the two buttons available allow the user to `cancel`, which resets the application to its initialised state, or to `retry`, which allows the application to re-test the environment noise. If the test passes either with or without additional filtering, the user can leave the test and begin their recording, as the `AudioProvider` enables the `record` button by setting `testPhase` to `'passed'`. As previously mentioned, the `AudioWorkletProcessors` run on separate threads from the main application thread and receive messages via `port.onmessage` sent from `port.postMessage`.

6.2 Environment Test

The `environmentTesting` utility module holds the logic for capturing audio for environmental noise assessment (see code appendix). To implement this feature, a 4-second capture loop runs, after which the captured audio samples are assessed for level, variability, and frequency distribution. The audio samples are also converted from the time domain to the frequency domain, which allows the application to produce a noise power spectrum for use with the Wiener filter. The 4-second capture loop is divided into 25 ms frame intervals, yielding 160 frames of data. During each of the 160 frames, the Web Audio API method `getFloatTimeDomainData` is called. This method fills an array called `timeBuffer` with the most recent 2048 sample values, roughly 42.7 ms worth (assuming a 48 kHz sample rate and an `fftsize` of 2048), by collecting samples from a circular buffer implemented by the `AnalyserNode` of the Web Audio API (not part of the application code). A sleep function is then called to pause the capture loop for 25 ms, allowing the browser, via the Web Audio API, to collect more data, after which `getFloatTimeDomainData` calls for another 2048 samples, however, given only 25 ms have passed, the most recent 2048 samples will only contain around 17.7 ms of new samples (42.7 ms - 25 ms). This use of overlapping frames provides a smoother, continuous picture of how the audio signal changes over time. Another utility function, `calculateRmsAmplitude`, is also called and returns a single root-mean-square amplitude for all samples captured during that frame, with the value stored in a new array for later use in the environment test. Finally, the capture loop also implements a bespoke `floatFrequencyData` method that converts time-

domain data to frequency-domain data. Initially, the Web Audio API `getFloatFrequencyData` method performed this task; however, during the implementation of the Wiener filter, it became apparent that discrepancies existed between the noise-spectrum power values produced by the environment test for the noise profile and those of the live signal passing through the Wiener filter, resulting in incorrect gain reduction (the Web Audio API returns dB-scaled, smoothed values rather than the linear power values the Wiener filter operates on). Given that the Wiener filter compares a desired signal with an undesired signal, it is vital that the two signals be properly scaled. To overcome this issue, the FFT implementation used by the Wiener filter was replicated in a main-thread function called `floatFrequencyData`, ensuring both use identical windowing, normalisation, and power computation. It should be noted that methods cannot be easily shared between main-thread modules and `AudioWorkletProcessor` scripts, so copying the code made more sense in this instance.

Once the capture cycle is complete, the average noise-spectrum power is computed from the 160 captured frames. There are several stages to this process, and it employs complex mathematics, the details of which are beyond the scope of this paper. In broad terms, the array `noiseSpectrumPowerAvg` stores the power for 1025 frequency bins per captured frame, where each bin spans roughly 23.4 Hz. This value is calculated by dividing the sample rate (48 kHz) by the FFT size (2048). The power value stored in the array represents the summed power across all frequencies within that bandwidth. When assigning frequency bin values to the given array, we use only power values from bin 0 to bin 1024, as beyond this point, all values are redundant for real-valued inputs. Real-valued numbers are ordinary real numbers that do not contain a complex part, such as audio samples, which are just amplitude measurements at a point in time. During the capture period, the power values for each bin are summed. The first action after the capture loop completes is to average these values by dividing each summed value by the total number of frames (160 in this instance). A for loop mutates the `noiseSpectrumAvgPower` array by replacing the summed values with the averaged values. When a noise filter is required, the `noiseSpectrumAvgPower` array is returned as the representative noise profile for the captured environment.

Before setting the noise filter, a series of time-domain calculations are performed to determine the median RMS (root-mean-squared) noise level and the RMS noise levels at the 10th and 90th percentiles. To determine the median noise level, the values stored in `rmsFrames` are converted from amplitude to decibel units using a utility function called `amplitudeToDb`, and the results are stored in a new array called `rmsFramesDbFS`. This conversion isn't strictly necessary, but it allows the use of decibel values when setting thresholds for checks completed in this environment test, which, in turn, makes it easier to determine whether performance is correct or incorrect. The `rmsFramesDbFS` values are then passed through a percentile function that takes two arguments: the array and the percentile value to be calculated. The function sorts the array's RMS values in ascending order, then returns the value at the median (50th percentile) index. The median noise level is compared to a maximum noise constant of -40 dBFS; if it exceeds this value, the noise is deemed too loud, and the test fails. If the value is below -65 dBFS, then it is deemed quiet enough to be ignored, and the test passes with no filter applied. The same method could be applied to the noise level instability check by passing the array and the values 10 for the 10th percentile and 90 for the 90th percentile; however, it is important to consider the amount of time each frame represents. As discussed, each captured frame is separated by 25 ms. A sudden loud spike in level that lasts 200 ms would occupy the frames above the 95th percentile, or, in other words, would be in the last 8 frames ($8 \times 25 = 200\text{ms}$) once the percentile sorted array function is applied. To improve this, the average of 8 frames is combined into a single block, yielding 20 blocks in total, and each block is pushed into a new array called `blockRms`. The array values are then mapped to decibel units using the `amplitudeToDb` method, and the new array is passed to the same percentile function, from which the p10 and p90 values are derived. The instability of the captured audio is determined by the spread between p10 and p90; anything over 6 dB is deemed too unstable.

If the environmental noise is deemed significant enough, not too unstable, and not too loud, the

noise frequency distribution is assessed to determine whether it is dominant on either side of the typical respiratory frequency band. The respiratory frequency band is set at 200 Hz to 2000 Hz; the respiratory band is generally considered wider than this. However, this allows overlap, and the Wiener filter only attenuates frequency bins where the noise power exceeds the signal power; it does not reduce frequencies where the live signal is already stronger than the noise. With the band set, the environment test needs to determine which frequency bin in the `noiseSpectrumAvgPower` array corresponds to 200 Hz and which to 2000 Hz. This is determined by the following formula:

```
1 freqBinNumber = Math.floor((frequency / Nyquist frequency) * (fftsize / 2 + 1))
```

Figure 8: Frequency Bin Number Calculation

This is because the `noiseSpectrumAvgPower` array has 1025 frequency bins, spanning 0 to 24 kHz, or the Nyquist frequency, which is half the sample rate. Once the frequency bin numbers are determined, a for loop iterates over all power values, summing them for the total whilst also summing them for the lower and upper bin numbers. From these values, two ratios are derived: if either the low-frequency ratio exceeds 40% or the high-frequency ratio exceeds 5% of the total noise energy, the noise filter is applied. The type of noise to be filtered by such a system is typically low-frequency-dominated, such as traffic, HVAC systems, or rumble.

7 Results and Discussion

7.1 The Product

The finished product is a working web application that provides the user with registration and login functionality. Once logged in, the user can access any connected microphones. Once the microphone is running, the user can test their background for suitability and, if certain criteria are met, record and upload an audio sample. The system automatically uploads 3 audio files, raw, filtered and main. The raw file is taken from the start of the audio graph and only contains the mono mix processing. The filtered audio is the output after it passes through the low-pass and high-pass filters. If Wiener filtering is applied, the main file will include this processing; otherwise, the main and filtered files will be the same. It should be noted that the Wiener processor introduces a 2048-sample shift in the main audio file. If a local version of VoxLab is running, when uploading the files to MongoDB, the main file will also be uploaded to VoxLab. Additional endpoints, not accessible via the UI, allow retrieval of all audio records and individual audio file streams.

7.2 Audio Consistency

For the application to function correctly, it is vital that the audio metrics gathered by the Web Audio API analyser node are accurate. To test this element of the application, two parallel recordings were made from the same microphone input. One recording was captured via Audacity, a free, open-source, cross-platform audio recording and editing program, and the other via the respiratory web application, which was subsequently stored in the database for retrieval. A fan recording played back via an external device provided the sound source. Two key metrics for the application to function are the root-mean-square (RMS) and peak levels of the audio signal. The images below show the application's UI alongside two Audacity readings. The first reading is the RMS of the audio captured directly by Audacity, the second is the RMS of the raw audio retrieved from the respiratory audio application. These three images

highlight the consistency between the UI display, the raw audio captured by the application, and the corroborated audio in Audacity.

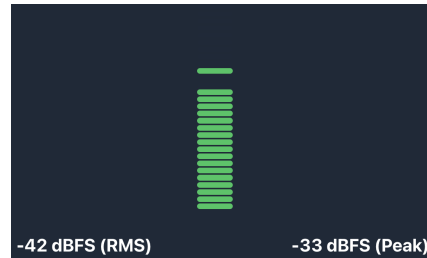


Figure 9: Audio Recorder UI

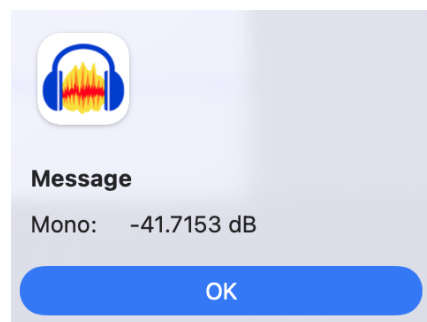


Figure 10: Audacity RMS

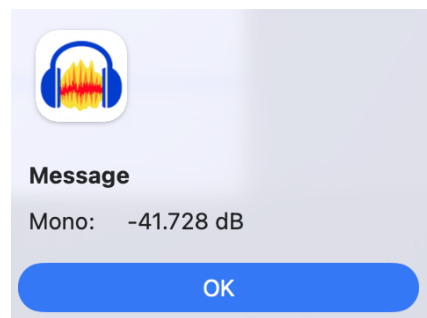


Figure 11: Audacity RMS

The UI value represents a single moment in time, so exact agreement with a whole-file RMS measurement is not expected; rather, the UI value should fall within the range observed across the recording. Below is a portion of the logs captured by the application, which display key metrics for the audio recording. One of the metrics provided is `signalNoiseLevel`, which is the RMS of the entire audio signal captured during the environment test. The value of 41.381 dB is within 0.5 dB of the Audacity values.

```

1 "metricDetails": {
2   "CLEAN_SIGNAL_DB_MAX": -55,
3   "HIGH_FREQ_RATIO_MAX": 0.05,
4   "LOW_FREQ_RATIO_MAX": 0.4,
5   "peak": -31.566228423486766,
6   "signalHighFreqRatio": 0.032994492229928805,
7   "signalLowFreqRatio": 0.7259044110124064,
8   "signalNoiseLevel": -41.38164442321953
9 },

```

Figure 12: Recording Logs

These logs also report a maximum peak level of -31.566 dB during recording. A simple method for extracting peak levels from audio recordings is to use the Audacity “Amplify” plugin, which detects the highest peak and displays the required adjustment to set it to 0 dBFS. The images below show the two readings for the Audacity-captured audio and the raw application audio.

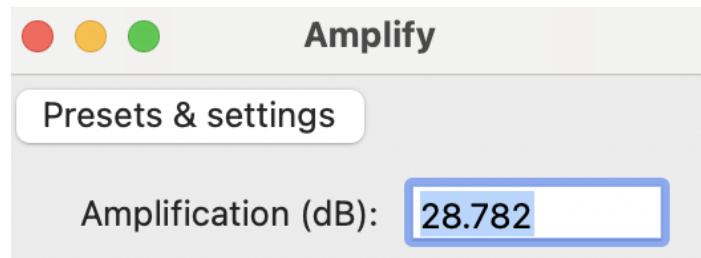


Figure 13: Audacity Peak Level

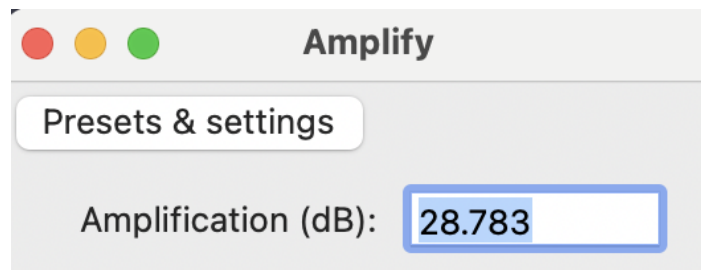


Figure 14: Raw Peak Level

These images highlight a 2.8 dB difference between the environment test measured peak level and the audio recording peak level. Whilst this is not insignificant, all key logic decisions made within the application use the RMS value, except for clipping safety checks. Within the environment test, a conservative clipping guard threshold is set at -6 dBFS, so if this 2.8 dB difference were to occur consistently, the application would still flag potential clipping before the signal reaches 0 dBFS, preventing actual digital clipping.

Overall, these tests show that the application’s calculations for both logic and user feedback are usable and accurately reflect the captured audio.

The first processing completed by the audio pipeline is to apply low-pass and high-pass filtering to ensure focus on the respiratory audio band. The low-pass filter cut-off frequency is set to 3500 Hz, and the high-pass filter cut-off frequency is set to 120 Hz. The chart below demonstrates the application of both filters. The chart shows the frequency spectrum of the raw audio prior to filtering, and the audio after filtering but before any additional noise filtering. The orange graph highlights the gradual

reduction in the level of all frequencies above 3.5 kHz and below 120 Hz, demonstrating the expected behaviour of the applied filters.

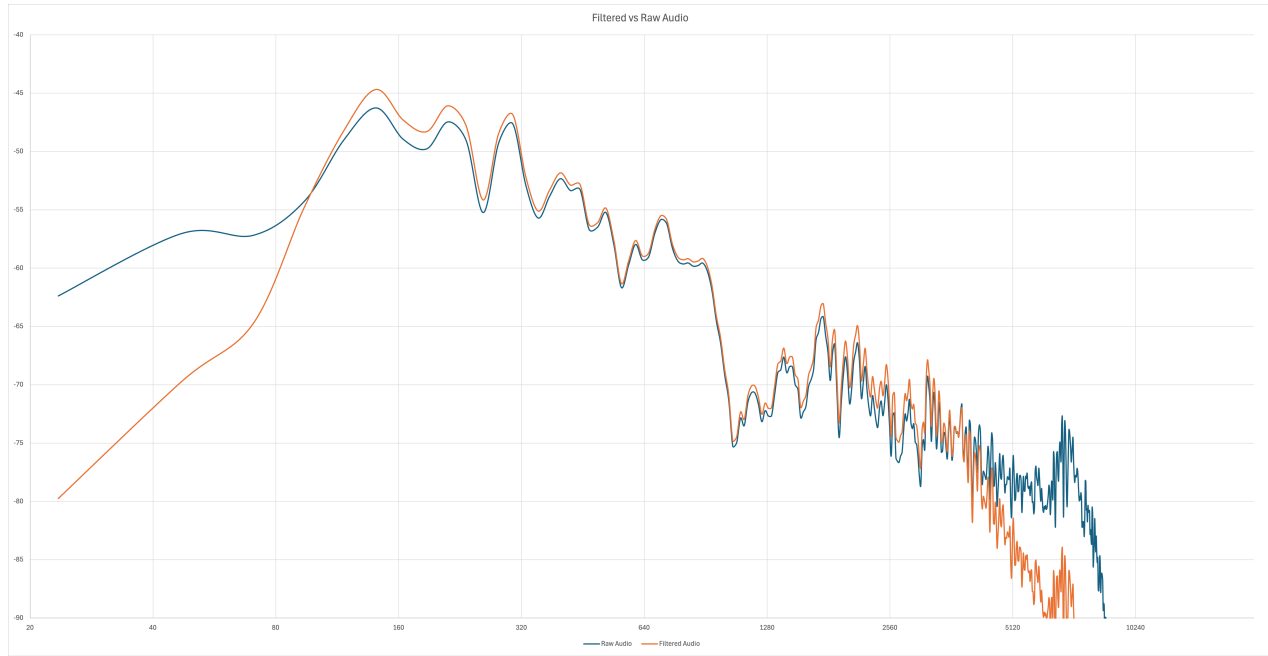


Figure 15: Raw Audio vs Filtered Audio

The final audio consistency check compares the noise profile data captured by the environment test with the raw audio spectrum of the noise it represents. This data is used by the Wiener filter to determine whether to reduce the levels of certain frequencies in the final captured audio by comparing the recorded signal to this noise profile. Both plots have the same shape; however, the noise profile is roughly 2 dB quieter than the filtered audio for the most part. The noise profile generation step and the audio recording occur one after the other, and discrepancies may result from slight changes in background noise over a short time period. A 2 dB difference will result in a noise estimate that is lower than the true noise floor, meaning the Wiener filter is conservative; it preserves signal at the cost of marginally more residual noise, whereas the opposite error would risk cancelling respiratory signal content.

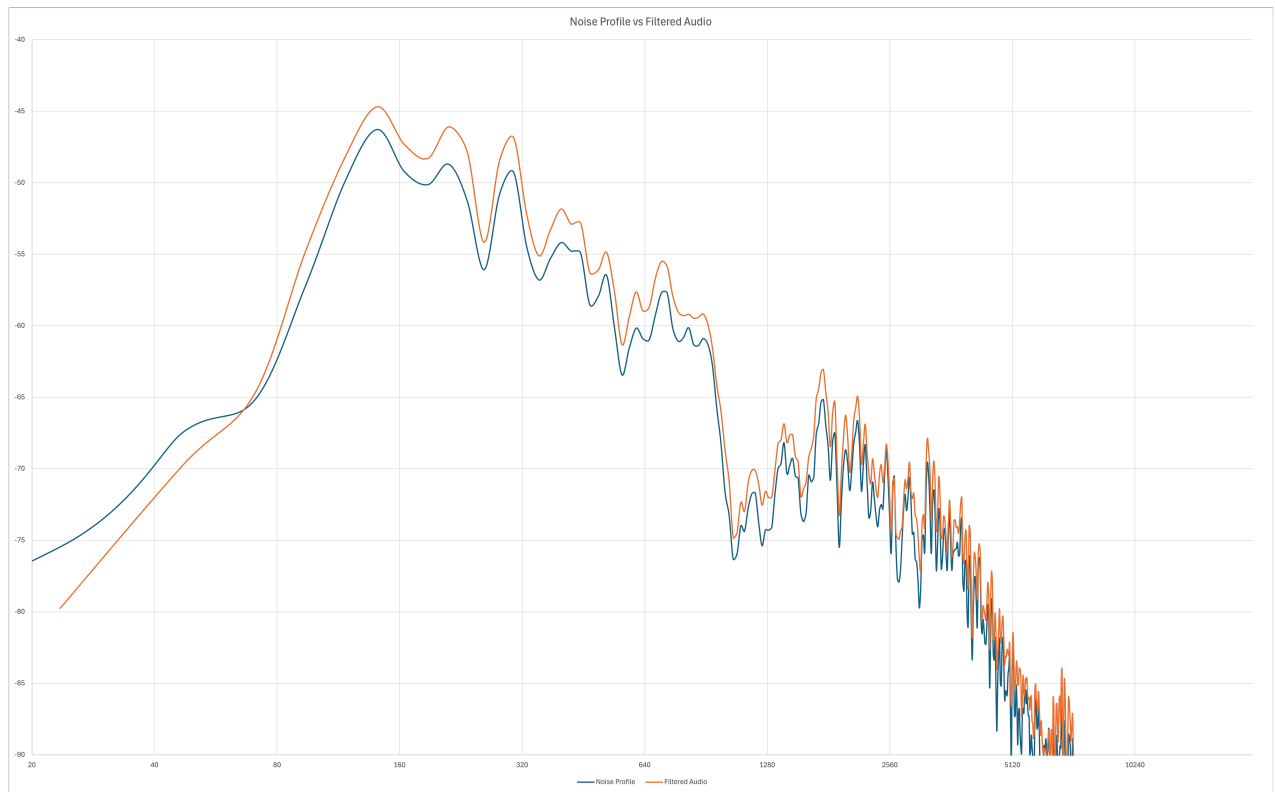


Figure 16: Filtered Audio vs Noise Profile

7.3 Environment Test

To assess the pass/fail logic of the environmental noise test, a series of different sound sources playing at varying levels will cause the system to pass, fail or pass with a filter applied. Each test generates a log captured and stored server-side, with the full logs available in the appendix. The following tests demonstrate the behaviour of the environment test:

- Static noise above the maximum threshold - Fail
- Static noise below the clean threshold - Pass
- Static noise above the clean threshold and below the maximum threshold - Pass with filter applied
- Variable noise above the clean threshold and below the maximum threshold - Fail
- Variable noise below the clean threshold - Pass
- Noise band-limited to the respiratory frequency band - Fail

Here are the key metrics for each test, extracted from the logs, along with an image of the audio waveform captured by Audacity during the test.

7.3.1 Static noise above maximum threshold - Fail

```
10 {
11   "level": "info",
12   "message": "-----ENVIRONMENT TEST STARTED -----",
13   "meta": {
14     "test": "TEST-1776803806599-ENV-00001"
15   },
16   "source": "client",
17   "timestamp": "2026-04-21T20:38:13.001Z"
18 },
19 {
20   "level": "info",
21   "message": "FAIL - NOISE TOO LOUD",
22   "meta": {
23     "metricDetails": {
24       "NOISE_DB_MAX": -40,
25       "avgNoiseDb": -32.35587053819333,
26       "peak": -24.43481182033827
27     },
28     "testId": "TEST-1776803806599-ENV-00001"
29   },
30   "source": "client",
31   "timestamp": "2026-04-21T20:38:17.310Z"
32 },
```

Figure 17: Static Noise Test Fail

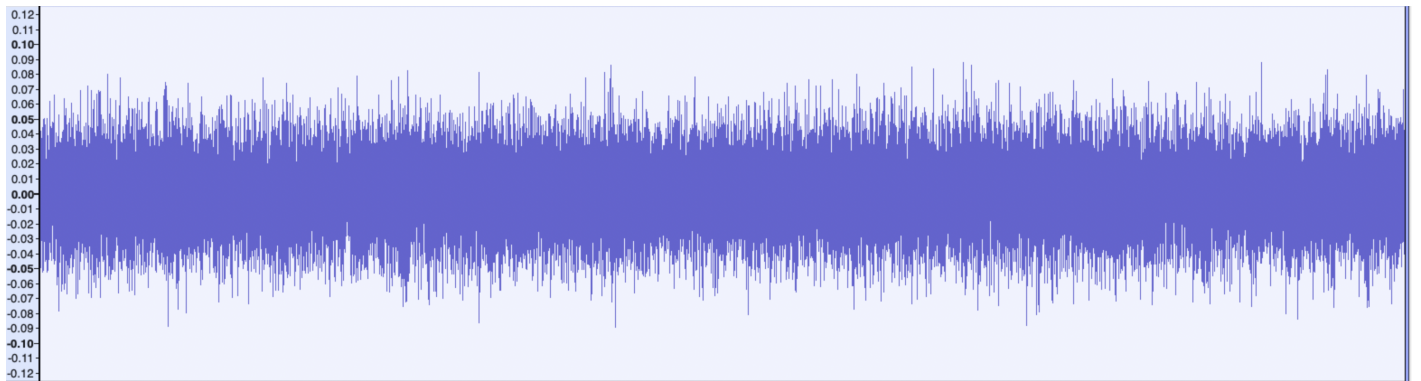


Figure 18: Loud static noise

7.3.2 Static noise below the clean threshold - Pass

```
33 {  
34   "level": "info",  
35   "message": "-----ENVIRONMENT TEST STARTED -----",  
36   "meta": {  
37     "test": "TEST-1776803806599-ENV-00002"  
38   },  
39   "source": "client",  
40   "timestamp": "2026-04-21T20:40:53.145Z"  
41 },  
42 {  
43   "level": "info",  
44   "message": "PASS - NO FILTER REQUIRED",  
45   "meta": {  
46     "metrics": {  
47       "CLEAN_DB_THRESHOLD": -55,  
48       "medianNoiseDbLevel": -66.7082874816754,  
49       "peak": -58.675841616299195  
50     },  
51     "testId": "TEST-1776803806599-ENV-00002"  
52   },  
53   "source": "client",  
54   "timestamp": "2026-04-21T20:40:57.460Z"  
55 }
```

Figure 19: Static Noise Test Pass

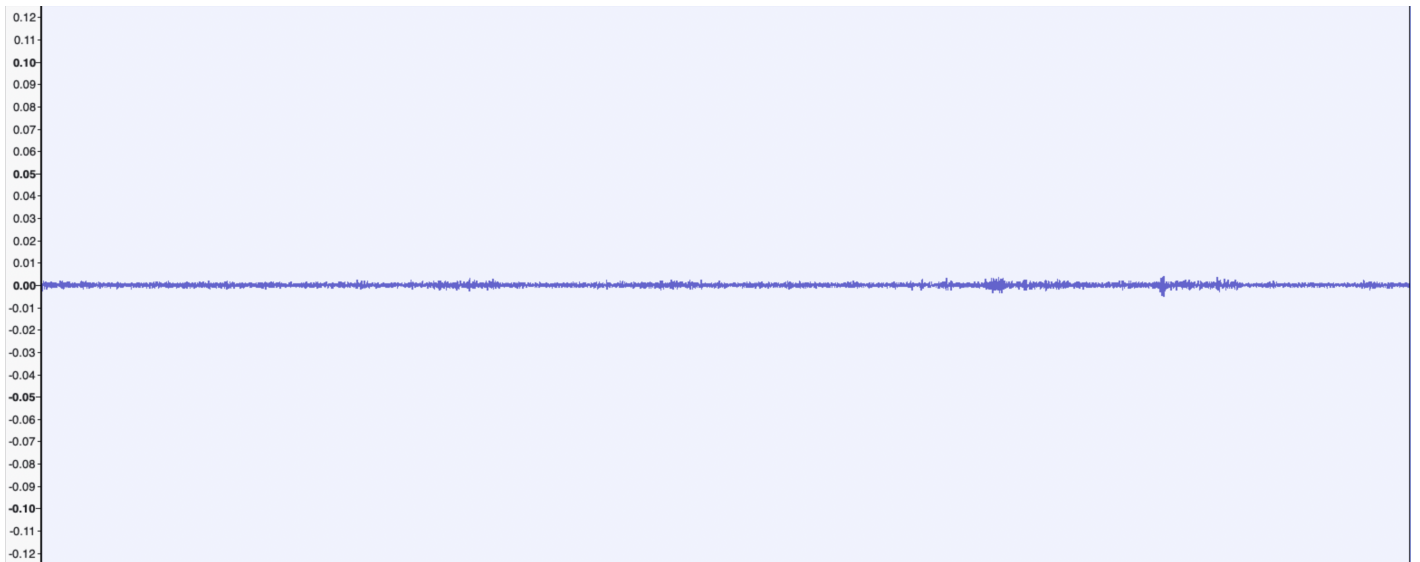


Figure 20: Quiet static noise

7.3.3 Static noise above the clean threshold and below the maximum threshold - Pass with filter applied

```
56 {
57   "level": "info",
58   "message": "-----TEST STARTED-----",
59   "meta": {
60     "test": "TEST-1776804179060"
61   },
62   "source": "client",
63   "timestamp": "2026-04-21T20:42:59.060Z"
64 },
65 {
66   "level": "info",
67   "message": "-----ENVIRONMENT TEST STARTED -----",
68   "meta": {
69     "test": "TEST-1776804179060-ENV-00001"
70   },
71   "source": "client",
72   "timestamp": "2026-04-21T20:43:25.388Z"
73 },
74 {
75   "level": "info",
76   "message": "PASS - NOISE PROFILE CREATED",
77   "meta": {
78     "metricDetails": {
79       "CLEAN_SIGNAL_DB_MAX": -55,
80       "HIGH_FREQ_RATIO_MAX": 0.05,
81       "LOW_FREQ_RATIO_MAX": 0.4,
82       "peak": -34.4865994605322,
83       "signalHighFreqRatio": 0.04309872456448348,
84       "signalLowFreqRatio": 0.6761624750553888,
85       "signalNoiseLevel": -45.1844794752666
86     },
87     "testId": "TEST-1776804179060-ENV-00001"
88   },
89   "source": "client",
90   "timestamp": "2026-04-21T20:43:29.710Z"
91 },
```

Figure 21: Static Noise Pass - Noise Profile Created



Figure 22: Static noise

7.3.4 Variable noise above the clean threshold and below the maximum threshold - Fail

```
92 {
93   "level": "info",
94   "message": "-----TEST STARTED-----",
95   "meta": {
96     "test": "TEST-1776804272936"
97   },
98   "source": "client",
99   "timestamp": "2026-04-21T20:44:32.936Z"
100 },
101 {
102   "level": "info",
103   "message": "-----ENVIRONMENT TEST STARTED -----",
104   "meta": {
105     "test": "TEST-1776804272936-ENV-00001"
106   },
107   "source": "client",
108   "timestamp": "2026-04-21T20:45:09.084Z"
109 },
110 {
111   "level": "info",
112   "message": "FAIL - NOISE TOO UNSTABLE",
113   "meta": {
114     "metricDetails": {
115       "RMS_INSTABILITY_DB_MAX": 6,
116       "instabilityInDb": 14.15853804513084,
117       "peak": -51.17475603257455
118     },
119     "testId": "TEST-1776804272936-ENV-00001"
120   },
121   "source": "client",
122   "timestamp": "2026-04-21T20:45:13.388Z"
123 },
124 {
125   "level": "info",
126   "message": "-----ENVIRONMENT TEST COMPLETE-----",
127   "meta": {
128     "test": "TEST-1776804272936-ENV-00001"
129   },
130   "source": "client",
131   "timestamp": "2026-04-21T20:45:13.389Z"
132 },
133 {
134   "level": "info",
135   "message": "-----ENVIRONMENT TEST FAILED-----",
136   "meta": {
137     "reason": "noise too unstable",
138     "test": "TEST-1776804272936-ENV-00001"
139   },
140   "source": "client",
141   "timestamp": "2026-04-21T20:45:13.389Z"
142 },
```

Figure 23: Variable Noise Fail



Figure 24: Variable noise fail

7.3.5 Variable noise below the clean threshold - Pass

```

143 {
144   "level": "info",
145   "message": "-----ENVIRONMENT TEST STARTED -----",
146   "meta": {
147     "test": "TEST-1776804272936-ENV-00002"
148   },
149   "source": "client",
150   "timestamp": "2026-04-21T20:46:15.766Z"
151 },
152 {
153   "level": "info",
154   "message": "PASS - NO FILTER REQUIRED",
155   "meta": {
156     "metrics": {
157       "CLEAN_DB_THRESHOLD": -55,
158       "medianNoiseDbLevel": -61.207239254694514,
159       "peak": -47.745125877568725
160     },
161   }
162 }

```

Figure 25: Quiet Variable Noise Pass



Figure 26: Quiet variable noise pass

7.3.6 Noise band-limited to the respiratory frequency band - Fail

```

163 {
164   "level": "info",
165   "message": "-----ENVIRONMENT TEST STARTED -----",
166   "meta": {
167     "test": "TEST-1776804436167-ENV-00001"
168   },
169   "source": "client",
170   "timestamp": "2026-04-21T20:47:49.416Z"
171 },
172 {
173   "level": "info",
174   "message": "FAIL - NOISE IN RESPIRATORY BAND",
175   "meta": {
176     "allMetrics": {
177       "CLEAN_SIGNAL_DB_MAX": -55,
178       "HIGH_FREQ_RATIO_MAX": 0.05,
179       "LOW_FREQ_RATIO_MAX": 0.4,
180       "RMS_INSTABILITY_DB_MAX": 6,
181       "highFrequencyRatio": 0.0030442615391076118,
182       "lowFrequencyRatio": 0.16869228767819716,
183       "medianNoiseDbLevel": -42.177041581451036,
184       "noiseInstability": 0.9285774473535113,
185       "peak": -31.680728422845654
186     },
187   },
188 }

```

Figure 27: Band-limited noise fail



Figure 28: Band-limited noise fail

7.4 Wiener Filter

To test and demonstrate the application of the Wiener filter, the same fan noise used for the environment test and audio consistency test will cause the environment test to pass with the filter applied. Once the filter is applied, the user will start a recording, cough at roughly 1-second intervals, and upload the completed recording to MongoDB. Below are Audacity-generated spectrograms of the pre- and post-Wiener-filter audio files, along with a difference spectrogram. To generate a difference spectrogram, the post-Wiener filter waveform is shifted by 2048 samples, due to the sample shift introduced by the filtering process, and inverted. A new audio file is then generated by combining the pre- and post-Wiener-filter waveforms, leaving only the difference between them.

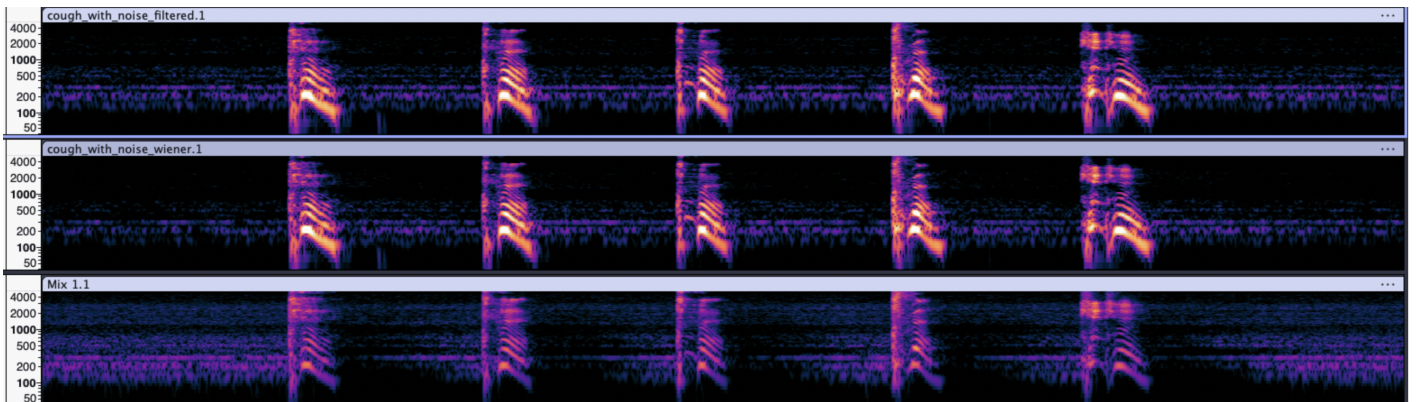


Figure 29: Spectrograms - Top: pre noise reduction, middle: post noise reduction, bottom: difference

Spectrograms combine time, frequency, and signal level in a single graph. The x- and y-axes are for time and frequency, with signal level represented by colour intensity. Therefore, as highlighted by the slightly darker sections of the post-Wiener spectrogram between each cough, some of the noise has been reduced; however, the reduction is not significant. This is also confirmed by the difference spectrogram. The images below are the same pre- and post-spectrograms, except their dB ranges are now set to different values. With the pre-Wiener set at 47 dB and the post-Wiener set at 50 dB, the images now

look similar, suggesting a reduction of about 3 dB. The difference spectrogram also highlights that some of the respiratory audio has also changed as a result of the Wiener filtering.

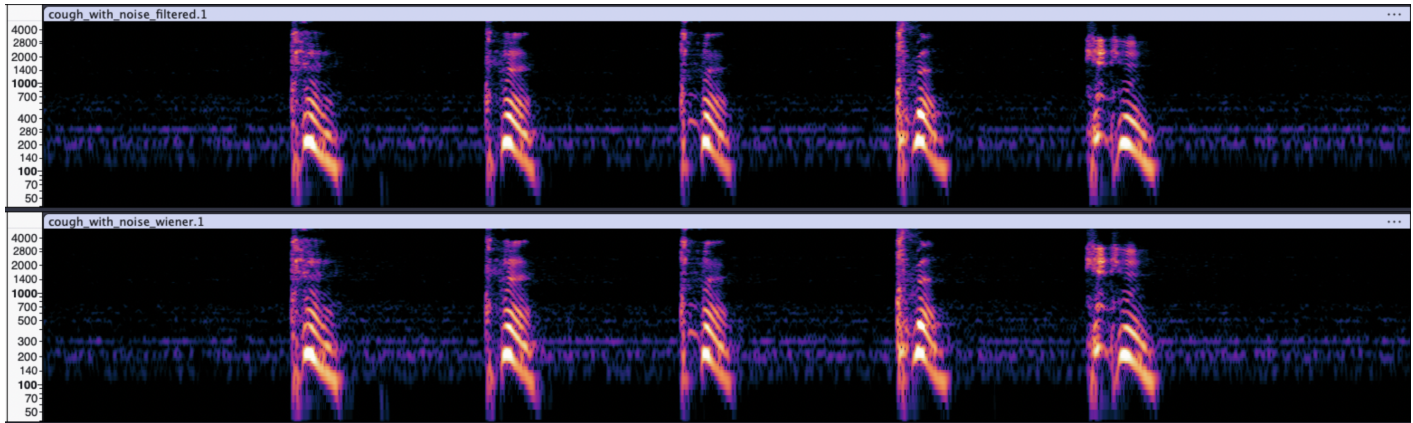


Figure 30: Spectrograms - Top: pre noise reduction, bottom: post noise reduction

8 Conclusions and Recommendations

The main outcome of this work is a working web-based application that allows users to record and store audio for future retrieval. More specifically, the project has produced a guided microphone setup workflow, real-time bandpass filtering tuned to respiratory frequencies, an environment testing system with noise profiling, multiple output files at each processing stage, and authenticated storage with streaming retrieval. The principal results demonstrate that the Web Audio API is a flexible and powerful tool for developing such an application, and analysis confirmed that accurate measurements of audio recordings are achievable and that real-time decisions and workflows can utilise these measurements to assist users with audio recording. All of the stated aims and objectives have been met. An audio workflow that guides the user through the process was designed and implemented. A state machine with dedicated UI components for each stage of the process allows the user to successfully select and start a microphone, assess background noise, record and then submit multiple audio files. The audio pipeline includes audio filters to attenuate frequencies above and below the respiratory band. The environment test captures a 4-second snippet of background noise and assesses the environment's suitability for recording. The test results and logs demonstrate the application's performance against these objectives and that the metrics used to determine the application behaviour are relevant and non-trivial. These tests also highlight that a successful recording submission generates multiple audio files from various stages of the audio pipeline, with each file stored in MongoDB and retrievable via an HTTP client, such as Bruno. Whilst the implementation of a Wiener filter meets one objective, its use in this application is limited by the minimal noise reduction it can achieve, and further development is required if it is to be of any use to this pipeline.

This work is a good starting point for future web-based audio capture systems and demonstrates a successful implementation of the Web Audio API using bespoke AudioWorkletProcessors, further increasing the API's functionality, all within the widely used MERN development stack. In addition to the stated aims and objectives, this service is integrated with a vocal analysis tool. Also, using the React Context API to manage complex state across components was vital, as it significantly reduced component interface complexity and ensured a single source of truth for certain data.

There are four key areas that require further development to meet the needs outlined in the literature review regarding poor audio recording techniques. Firstly, there is currently no signal-to-noise ratio

check. This was discussed as part of the design process; however, time constraints prevented a successful implementation. To ensure users are not recording in sub-par environments, it's vital that the captured signal is as loud as possible, without clipping, and that there is sufficient dynamic range between the desired and undesired audio. A major barrier to implementing such a feature is the inability to control system settings from within a web application. Whilst an application can provide gain control to prevent clipping during processing, if clipping occurs at a hardware level, there is not much that can be done about it after this point. Conversely, if the system audio is too low, then excessive gain applied by the application may result in poor audio quality. It should be noted that without some form of gain staging, the environment test is unable to fully assist the user. Where the environmental noise is too loud, the user can simply lower their system audio until it reaches a level that satisfies the environment test logic, which is only useful to a point. Lowering the system audio may result in the desired signal being too quiet. A signal-to-noise ratio check would quickly highlight this and, if possible, guide the user to a satisfactory balance. Ultimately, spectral and dynamic separation between desired and undesired signals is key to producing clean audio recordings. This leads to the second area for future development, the Wiener filter or more specifically, noise reduction. In its current form, the Wiener filter does not add much to this application. Implementing such a tool requires highly complex digital signal processing and is really beyond the scope of this paper. The module could only be developed with AI assistance to the point that this author takes no real ownership of the code, and in hindsight, an "off-the-shelf" solution, such as RNNoise via WASM or server-side processing via Python libraries such as noisereduce, may have provided more value to a web development project. The module required extensive bugfixing, including identifying audible issues and using detailed logs to better understand the performance. Although the obvious solution to a poor recording environment is to move somewhere else, that may not always be possible, especially if the device itself is causing the noise (e.g., a computer fan). Therefore, improving the ability to reduce or remove noise without distorting the desired signal is vital to the success of such an application. The third area for future development is the ability to provide an audio recording assessment prior to submission. Even with assistance, poor recordings can still occur and preventing the submission of substandard recordings is the main purpose of an application such as this. Key metrics would include signal clipping or distortion, a low signal-to-noise ratio, and excessive low-frequency content, which suggest wind noise or proximity to the microphone. To prevent frustration, a grading of severity would help users submit audio that meets a minimum standard while also highlighting that the audio quality could be improved. Finally, introducing post-processing would allow certain defects to be fixed prior to submission. Gentle normalisation can increase the overall signal level, provided it does not introduce distortion. Where recordings have excessive low-frequency content, some form of additional filtering will improve the overall tonal quality of the audio.

Beyond the audio pipeline and the issues highlighted in the literature review, there are a number of key additions that would improve a service such as this. The ability to complete and set questionnaires in the application will help clinicians gather vital patient information to supplement any audio recordings. The integration of wearable technology can provide another layer of data to assist in diagnosing and supporting respiratory conditions. Being a web-based application also opens the possibility of converting it to a Progressive Web App for greater cross-compatibility and improved user access.

A Flow Diagrams

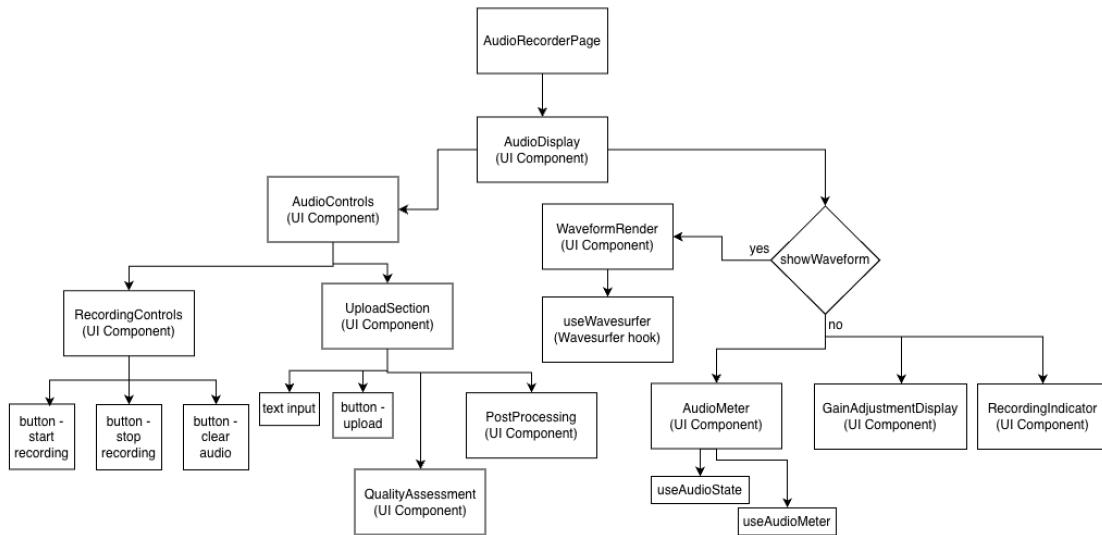


Figure 31: Flowchart - Audio Recorder

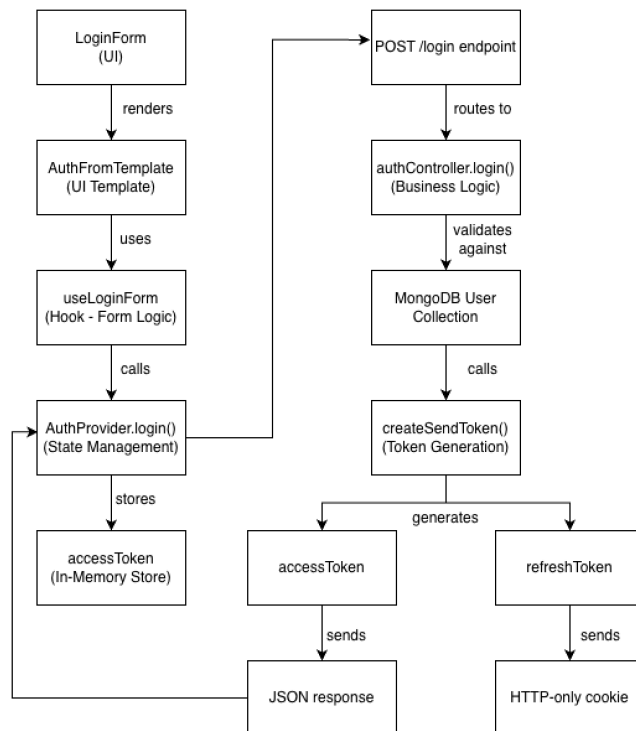


Figure 32: Flowchart - Auth

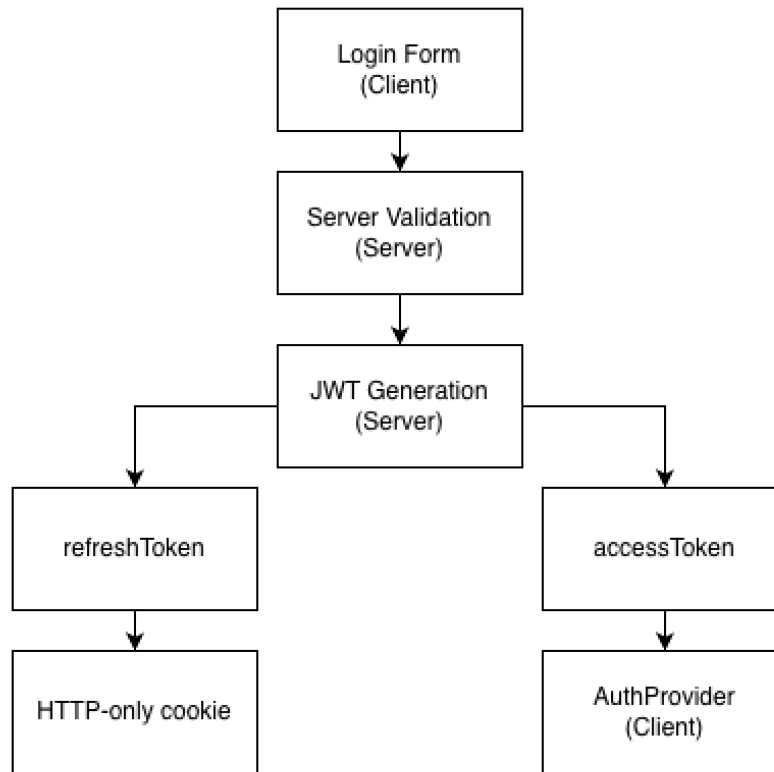


Figure 33: Flowchart - Login

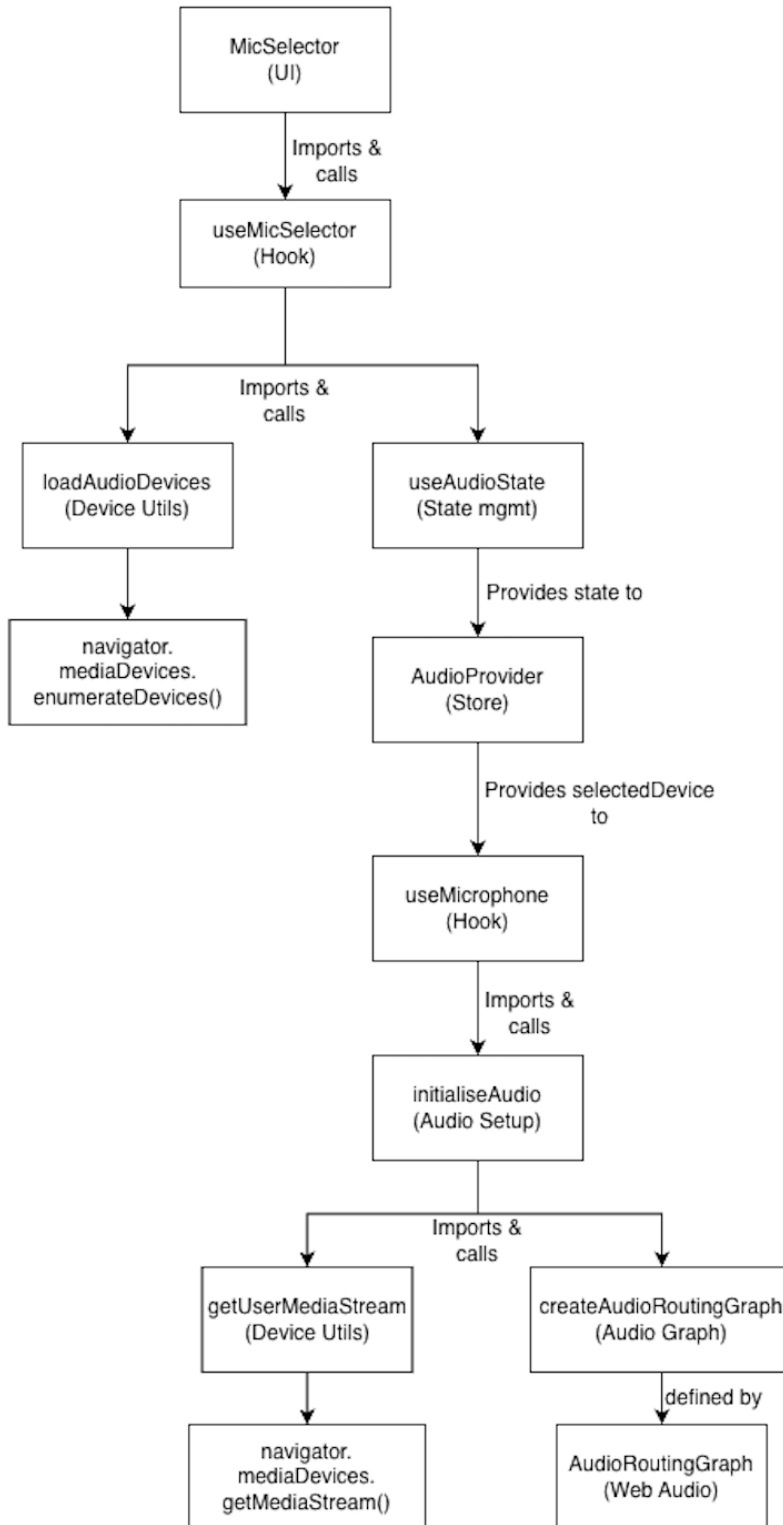


Figure 34: Flowchart - Mic Selector

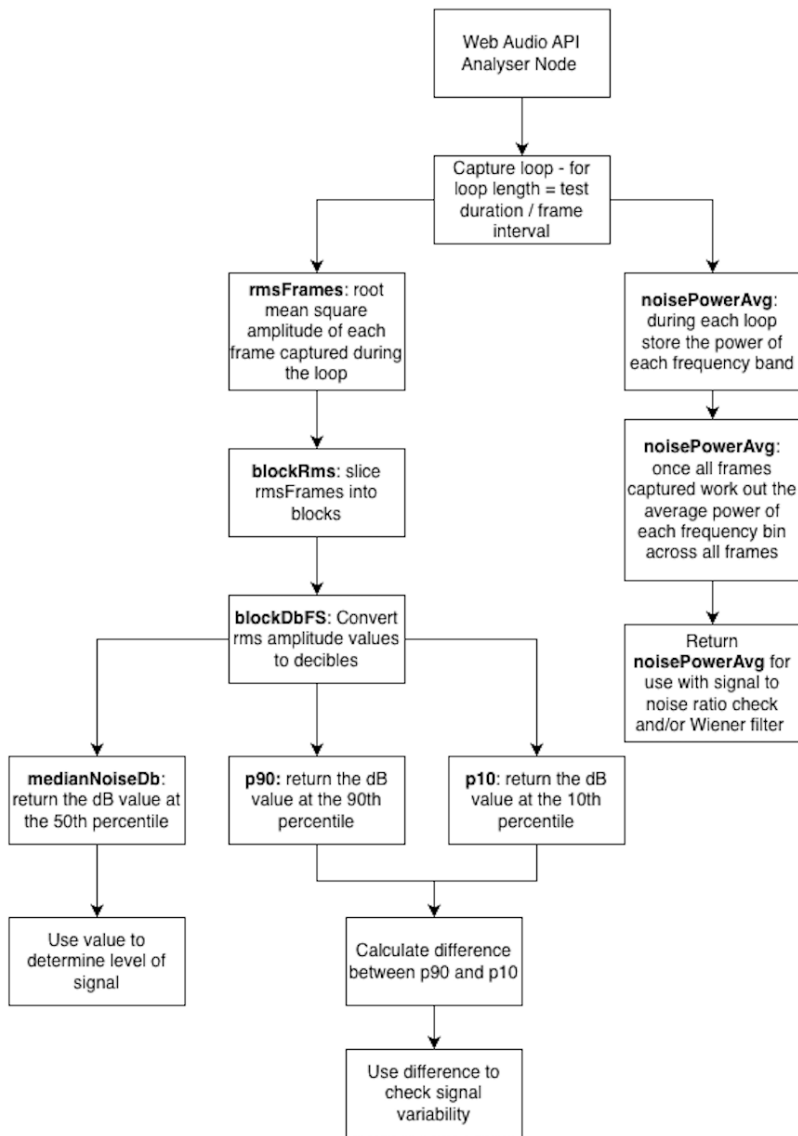


Figure 35: Flowchart - Capture Loop

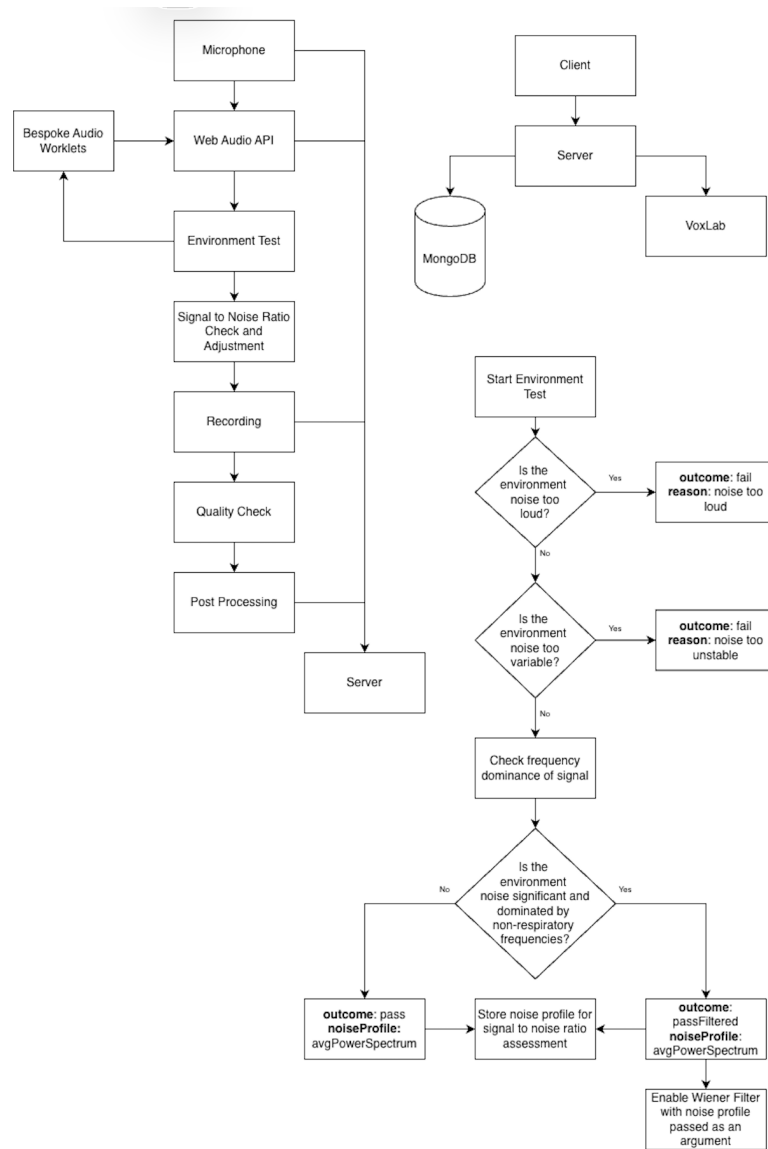


Figure 36: Flowcharts

B Patient User Stories

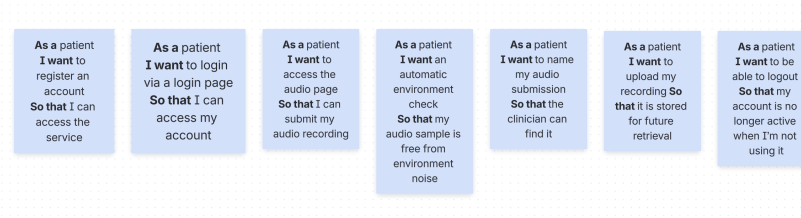


Figure 37: Basic Patient User Stories

C Code Snippets

```
2 import { useState, useRef, useCallback } from 'react';
3 import { AudioContext } from './AudioContext';
4 import { sleep, useMicrophone, type NoiseProfile } from '@features';
5 import { DB_DEFAULTS, MIN_DB_SPL_IDLE } from '@lib';
6 import { type AudioControllerState, type RecordedBlobs } from './types';
7 import { logger } from '@logging';
8 import { runEnvironmentTest } from '@features/audio/utils/environmentTesting';
9
10 export const AudioProvider = ({ children }: { children: React.ReactNode }) => {
11   const [isActive, setIsActive] = useState<boolean>(false);
12   const [audioDevices, setAudioDevices] = useState<Array<{ name: string; label: string }>>([]);
13   const [selectedDevice, setSelectedDevice] = useState<string>('');
14   const [amplitudeGain, setAmplitudeGain] = useState<number>(1);
15   const [isRecording, setIsRecording] = useState<boolean>(false);
16   const [uploadStart, setUploadStart] = useState<boolean>(false);
17   const [isUploading, setIsUploading] = useState<boolean>(false);
18   const [micSetupState, setMicSetupState] = useState<AudioControllerState>({
19     lifecycle: 'idle',
20     testPhase: 'none',
21   });
22   const [rawAudioDbSpl, setRawAudioDbSpl] = useState<number | undefined>(MIN_DB_SPL_IDLE);
23   const [rawAudioDbFsRms, setRawAudioDbFsRms] = useState<number | undefined>(DB_DEFAULTS.MIN_FS);
24   const [rawAudioDbFsPeak, setRawAudioDbFsPeak] = useState<number | undefined>(DB_DEFAULTS.MIN_FS);
25   const [filename, setFilename] = useState<string>('');
26   const [recordedUrl, setRecordedUrl] = useState<string>('');
27   const [recordedBlobs, setRecordedBlobs] = useState<RecordedBlobs>({
28     mainBlob: null,
29     filteredBlob: null,
30     rawBlob: null,
31     postProcessedBlob: null,
32   });
33   const [environmentTestNumber, setEnvironmentTestNumber] = useState<number>(1);
34   const [environmentTestState, setEnvironmentTestState] = useState<string | null>(null);
35   const [medianNoiseDb, setMedianNoiseDb] = useState<number | null>(null);
36   const testId = useRef<string>('');
37   const environmentTestId = useRef<string>('');
38
39   const mic = useMicrophone(selectedDevice, amplitudeGain);
40
41   /**
42    * when called resets audio state to initialised state
43    */
44   const resetAudioState = useCallback(() => {
45     logger.info('Reset audio state has been called', { test: testId.current });
46     setIsActive(false);
47     setIsRecording(false);
48     setAmplitudeGain(1);
49     setRawAudioDbFsRms(DB_DEFAULTS.MIN_FS);
50     setRawAudioDbFsPeak(DB_DEFAULTS.MIN_FS);
51     setRawAudioDbSpl(DB_DEFAULTS.MIN_FS);
52     setMicSetupState({ lifecycle: 'idle', testPhase: 'none' });
53     setEnvironmentTestNumber(1);
54
55     if (mic.wienerFilterNodeRef.current) {
56       mic.wienerFilterNodeRef.current.port.postMessage({ type: 'reset' });
57     }
58   }, [mic]);
59
60   const stopAll = () => {
61     logger.info('-----TEST STOPPED-----', {
62       test: testId.current,
63     });
64     resetAudioState();
65     mic.stopMic();
66     testId.current = '';
67     environmentTestId.current = '';
68   };
69
70   /**
71    * When called microphone starts and audio starts to stream updates setup state and mic active state
72    * @returns void
73    */
74   const startMicAndSetup = useCallback(async () => {
```

```

266 import { logger } from '@logging';
267 import {
268   BLOCK_SIZE,
269   CAPTURE_DURATION_MS,
270   CLEAN_DB_THRESHOLD,
271   FRAME_INTERVAL_MS,
272   HIGH_FREQ_CUTOFF_HZ,
273   HIGH_FREQ_RATIO_THRESHOLD,
274   LOW_FREQ_CUTOFF_HZ,
275   LOW_FREQ_RATIO_THRESHOLD,
276   MAX_ALLOWED_PEAK_DBFS,
277   NOISE_DB_MAX,
278   RMS_DB_VAR_MAX,
279 } from '@lib';
280 import { sleep } from './timing';
281 import { calculateRmsAmplitude } from './audioCalculations';
282 import { floatFrequencyData, amplitudeToDbFS } from './audioConversions';
283 import { type EnvironmentTestOutcome } from './types';
284
285 const percentile = (arr: number[], p: number): number => {
286   // if array length is 0 return out of function
287   if (arr.length === 0) return -1;
288   if (p < 0 || p > 100) return -1;
289
290   // copy array values and sort ascending
291   const sorted = [...arr].sort((a, b) => a - b);
292
293   // given a percentile value find the index value at that point e.g. for
294   // p = 50 -> 0.5 * (20 - 1) -> 0.5 * 19 = 9.5
295   const index = (p / 100) * (sorted.length - 1);
296
297   // this returns the greatest integer less than or equal to the index e.g. 9
298   const lower = Math.floor(index);
299
300   // this returns the smallest interger greater than or equal to the index e.g 10
301   const upper = Math.ceil(index);
302
303   return lower === upper ? sorted[lower] : (sorted[lower] + sorted[upper]) / 2;
304 };
305
306 export const runEnvironmentTest = async (analyser: AnalyserNode, audioContext: AudioContext, testId:
  string): Promise<EnvironmentTestOutcome> => {
307   if (!analyser || !audioContext) return { outcome: 'fail', reason: 'audio stream error' };
308
309   // extract fftSize from analyser to ensure value is correct
310   const fftSize = analyser.fftSize;
311
312   // setup buffers for loading samples with size set to fftSize
313   const timeBuffer = new Float32Array(fftSize);
314   const freqRe = new Float32Array(fftSize);
315   const freqIm = new Float32Array(fftSize);
316   // for storing the average power of each frequency bin of the captured environment noise
317   const noiseSpectrumAvgPower = new Float32Array(fftSize / 2 + 1);
318
319   const rmsFrames: number[] = [];
320
321   // number of frames captured during the test e.g. 160
322   const frameCount = Math.floor(CAPTURE_DURATION_MS / FRAME_INTERVAL_MS);
323
324   /** AUDIO SAMPLE CAPTURE LOOP */
325   // During the 4000 ms test, every 25 ms (see sleep function) a snapshot of the most recent
326   // ~42.7ms of audio data (2048 samples/ 48000 sample rate) is captured from the analyser
327   // causing an overlap where on every subsequent capture roughly 17.7 ms of samples were
328   // in the previous capture. Overlapping frames is intentional as it provides
329   // a smoother continuous picture of how the audio signal is changing over time
330   for (let i = 0; i < frameCount; i++) {
331     // fills the timeBuffer array with time domain audio samples (values between -1 and 1)
332     // that represent the audio signal
333     analyser.getFloatTimeDomainData(timeBuffer);
334
335     // Calculate the rms level taken from the time domain data during this frame iteration
336     const rms = calculateRmsAmplitude(timeBuffer);
337
338     // collect the rms level of the time domain data for each iteration
339     rmsFrames.push(rms);
340
341     // performs a Fast Fourier Transform to convert the time domain data from this iteration to

```

```

537 import { useRef, useCallback } from 'react';
538 import { cleanup, initialiseAudio } from '../utils';
539 import type { AudioRoutingGraphNodes } from '@app/providers';
540 import logger from '@logging/clientLogger';
541
542 export const useMicrophone = (selectedDevice: string, amplitudeGain: number) => {
543   const audioRoutingGraphNodesRef = useRef<AudioRoutingGraphNodes | undefined>(undefined);
544   const gainNodeRef = useRef<GainNode>(undefined);
545   const streamRef = useRef<MediaStream>(undefined);
546   const rawAudioAnalyserRef = useRef<AnalyserNode>(undefined);
547   const filteredAudioAnalyserRef = useRef<AnalyserNode>(undefined);
548   const recordedAudioAnalyserRef = useRef<AnalyserNode>(undefined);
549   const rawAudioDataArrayRef = useRef<Float32Array<ArrayBuffer>>(undefined);
550   const filteredAudioDataArrayRef = useRef<Float32Array<ArrayBuffer>>(undefined);
551   const recordedAudioDataArrayRef = useRef<Float32Array<ArrayBuffer>>(undefined);
552   const audioContextRef = useRef<AudioContext>(undefined);
553   const wienerFilterNodeRef = useRef<AudioWorkletNode>(undefined);
554
555   const startMic = useCallback(async () => {
556     try {
557       if (!selectedDevice) throw new Error('No device selected');
558       const audioRoutingGraphNodes = await initialiseAudio(selectedDevice);
559       if (!audioRoutingGraphNodes) throw new Error('Failed to initialise audio');
560
561       if (audioRoutingGraphNodes.audioContext.state === 'suspended') {
562         await audioRoutingGraphNodes.audioContext.resume();
563       }
564
565       audioRoutingGraphNodesRef.current = audioRoutingGraphNodes;
566       gainNodeRef.current = audioRoutingGraphNodes.gainNode;
567       audioContextRef.current = audioRoutingGraphNodes.audioContext;
568       wienerFilterNodeRef.current = audioRoutingGraphNodes.wienerFilterNode;
569
570       const currentGraph = audioRoutingGraphNodesRef.current!;
571
572       currentGraph.dcOffsetFilter.frequency.value = 5;
573       currentGraph.dcOffsetFilter.Q.value = 0.707;
574
575       currentGraph.highPassFilter.frequency.value = 120;
576       currentGraph.highPassFilter.Q.value = 0.707;
577
578       currentGraph.lowPassFilter.frequency.value = 3500;
579       currentGraph.lowPassFilter.Q.value = 0.707;
580
581       currentGraph.source.connect(currentGraph.rawAudioMonoMix);
582       currentGraph.rawAudioMonoMix.connect(currentGraph.rawAudioAnalyser);
583       currentGraph.rawAudioMonoMix.connect(currentGraph.rawAudioDestination);
584
585       currentGraph.source.connect(currentGraph.gainNode);
586       currentGraph.gainNode.connect(currentGraph.dcOffsetFilter);
587       currentGraph.dcOffsetFilter.connect(currentGraph.highPassFilter);
588       currentGraph.highPassFilter.connect(currentGraph.lowPassFilter);
589       currentGraph.lowPassFilter.connect(currentGraph.filteredAudioMonoMix);
590       currentGraph.filteredAudioMonoMix.connect(currentGraph.filteredAudioAnalyser);
591       currentGraph.filteredAudioMonoMix.connect(currentGraph.filteredAudioDestination);
592
593       currentGraph.lowPassFilter.connect(currentGraph.wienerFilterNode);
594       currentGraph.wienerFilterNode.connect(currentGraph.recordedAudioMonoMix);
595       currentGraph.recordedAudioMonoMix.connect(currentGraph.recordedAudioAnalyser);
596       currentGraph.recordedAudioMonoMix.connect(currentGraph.recordedAudioDestination);
597
598       streamRef.current = currentGraph.recordedAudioDestination.stream;
599       rawAudioAnalyserRef.current = currentGraph.rawAudioAnalyser;
600       filteredAudioAnalyserRef.current = currentGraph.filteredAudioAnalyser;
601       recordedAudioAnalyserRef.current = currentGraph.recordedAudioAnalyser;
602       rawAudioDataArrayRef.current = new Float32Array(rawAudioAnalyserRef.current.fftSize);
603       filteredAudioDataArrayRef.current = new Float32Array(filteredAudioAnalyserRef.current.fftSize);
604       recordedAudioDataArrayRef.current = new Float32Array(recordedAudioAnalyserRef.current.fftSize);
605     } catch (err) {
606       logger.error(err instanceof Error ? err.message : String(err), {
607         name: err instanceof Error ? err.name : String(err),
608       });
609     }
610   }, [selectedDevice]);
611
612   const stopMic = useCallback(() => {
613     if (audioRoutingGraphNodesRef.current) {

```

```

652 import { useAudioState } from '@app/providers';
653 import MicControls from './MicControls';
654 import MicReadyToTest from './MicReadyToTest';
655 import MicTestPrompt from './MicTestPrompt';
656 import MicTestingPanel from './MicTestingPanel';
657 import MicTestResult from './MicTestResult';
658 import MicReadyIndicator from './MicReadyIndicator';
659
660 const MicSetupPanel = () => {
661   const { micSetupState } = useAudioState();
662
663   switch (micSetupState.lifecycle) {
664     case 'idle':
665       return <MicControls />;
666     case 'micEnabled':
667       switch (micSetupState.testPhase) {
668         case 'none':
669           return <MicControls />;
670         }
671       return <MicReadyToTest />;
672     case 'testPrompt':
673       return <MicTestPrompt />;
674     case 'testing':
675       return <MicTestingPanel />;
676     case 'error':
677       return <MicTestResult />;
678     case 'ready':
679       switch (micSetupState.testPhase) {
680         case 'passedEnvironment':
681           return <MicTestResult />;
682         case 'passedSNR':
683           return <MicReadyIndicator />;
684         case 'passed':
685           return <MicControls />;
686       }
687   }
688 };
689
690 export default MicSetupPanel;

```

Figure 41: MicSetupPanel

References

- [1] J. B. Soriano, P. J. Kendrick, K. R. Paulson, V. Gupta, E. M. Abrams, R. A. Adedoyin, T. B. Adhikari, S. M. Advani, A. Agrawal, E. Ahmadian, F. Alahdab, S. M. Aljunid, K. A. Altirkawi, N. Alvis-Guzman, N. H. Anber, C. L. Andrei, M. Anjomshoa, F. Ansari, J. M. Antó, J. Arabloo, S. M. Athari, S. S. Athari, N. Awoke, A. Badawi, J. A. M. Banoub, D. A. Bennett, I. M. Bensenor, K. S. S. Berfield, R. S. Bernstein, K. Bhattacharyya, A. Bijani, M. Brauer, G. Bukhman, Z. A. Butt, L. A. Cámara, J. Car, J. J. Carrero, F. Carvalho, C. A. Castañeda-Orjuela, J.-Y. J. Choi, D. J. Christopher, A. J. Cohen, L. Dandona, R. Dandona, A. K. Dang, A. Daryani, B. de Courten, F. M. Demeke, G. T. Demoz, J.-W. De Neve, R. Desai, S. D. Dharmaratne, D. Diaz, A. Douiri, T. R. Driscoll, E. E. Duken, A. Eftekhari, H. Elkout, A. Y. Endries, I. Fadhil, A. Faro, F. Farzadfar, E. Fernandes, I. Filip, F. Fischer, M. Foroutan, M. Garcia-Gordillo, A. K. Gebre, K. B. Gebremedhin, G. G. Gebremeskel, K. E. Gezae, A. G. Ghoshal, P. S. Gill, R. F. Gillum, H. Goudarzi, Y. Guo, R. Gupta, G. B. Hailu, A. Hasanzadeh, H. Y. Hassen, S. I. Hay, C. L. Hoang, M. K. Hole, N. Horita, H. D. Hosgood, M. Hostiuc, M. Househ, O. S. Ilesanmi, M. D. Ilic, S. S. N. Irvani, S. M. S. Islam, M. Jakovljevic, A. A. Jamal, R. P. Jha, J. B. Jonas, Z. Kabir, A. Kasaeian, G. G. Kasahun, G. M. Kassa, A. T. Kefale, A. P. Kengne, Y. S. Khader, M. A. Khafaie, E. A. Khan, J. Khan, J. Khubchandani, Y.-E. Kim, Y. J. Kim, S. Kisa, A. Kisa, L. D. Knibbs, H. Komaki, P. A. Koul, A. Koyanagi, G. A. Kumar, Q. Lan, S. Lasrado, P. Lauriola, C. La Vecchia, T. T. Le, J. Leigh, M. Levi, S. Li, A. D. Lopez, P. A. Lotufo, F. Madotto, N. B. Mahotra, M. Majdan, A. Majeed, R. Malekzadeh, A. A. Mamun, N. Manafi, F. Manafi, L. G. Mantovani, B. G. Meharie, H. G. Meles, G. G. Meles, R. G. Menezes, T. Mestrovic, T. R. Miller, G. Mini, E. M. Mirrakhimov, B. Moazen, K. A. Mohammad, S. Mohammed, F. Mohebi, A. H. Mokdad, M. Molokhia, L. Monasta, M. Moradi, G. Moradi, L. Morawska, S. M. Mousavi, K. I. Musa, G. Mustafa, M. Naderi, M. Naghavi, G. Naik, S. Nair, V. Nangia, J. R. Nansseu, J. Nazari, D. E. Ndwandwe, R. I. Negoï, T. H. Nguyen, C. T. Nguyen, H. L. T. Nguyen, M. R. Nixon, R. Ofori-Asenso, F. A. Ogbo, A. T. Olagunju, T. O. Olagunju, E. Oren, J. R. Ortiz, M. O. Owolabi, M. P. A, S. Pakhale, A. Pana, S. Panda-Jonas, E.-K. Park, H. Q. Pham, M. J. Postma, H. Pourjafar, H. Poustchi, A. Radfar, A. Rafiei, F. Rahim, M. H. U. Rahman, M. A. Rahman, S. Rawaf, D. L. Rawaf, L. Rawal, R. C. Reiner Jr., M. B. Reitsma, L. Roever, L. Ronfani, E. M. Roro, G. Roshandel, K. E. Rudd, Y. D. Sabde, S. Sabour, B. Saddik, S. Safari, K. Saleem, A. M. Samy, M. M. Santric-Milicevic, B. P. Sao Jose, B. Sartorius, M. Satpathy, M. Savic, M. Sawhney, S. G. Sepanlou, M. A. Shaikh, A. Sheikh, M. Shigematsu, R. Shirkoohi, S. Si, S. Siabani, V. Singh, J. A. Singh, M. Soljak, R. Somayaji, M. Soofi, I. N. Soyiri, Y. M. Tefera, M.-H. Temsah, B. E. Tesfay, J. S. Thakur, A. T. Toma, M. Tortajada-Girbés, K. B. Tran, B. X. Tran, L. Tudor Car, I. Ullah, M. Vacante, P. R. Valdez, J. F. M. van Boven, T. J. Vasankari, Y. Veisani, F. S. Violante, G. R. Wagner, R. Westerman, C. D. A. Wolfe, D. Z. Wondafrash, A. B. Wondmienen, N. Yonemoto, S.-J. Yoon, Z. Zaidi, M. Zamani, H. J. Zar, Y. Zhang, and T. Vos, “Prevalence and attributable health burden of chronic respiratory diseases, 1990–2017: a systematic analysis for the Global Burden of Disease Study 2017,” *The Lancet Respiratory Medicine*, vol. 8, no. 6, pp. 585–596, Jun. 2020.
- [2] Y.-q. Hang, X. Piao, J. Wu, Q.-w. Jiang, and Y. Han, “The epidemiology of asthma and its attributable risk factors from 1990 to 2021: A systematic analysis based on the Global Burden of Disease Study 2021 and Mendelian Randomization Studies,” *Public Health*, vol. 243, p. 105731, Jun. 2025.
- [3] A. J. e. a. Ferrari, “Global incidence, prevalence, years lived with disability (YLDs), disability-adjusted life-years (DALYs), and healthy life expectancy (HALE) for 371 diseases and injuries in 204 countries and territories and 811 subnational locations, 1990–2021: a systematic analysis for

- the Global Burden of Disease Study 2021,” *The Lancet*, vol. 403, no. 10440, pp. 2133–2161, May 2024.
- [4] D. Adeloye, P. Song, Y. Zhu, H. Campbell, A. Sheikh, and I. Rudan, “Global, regional, and national prevalence of, and risk factors for, chronic obstructive pulmonary disease (COPD) in 2019: a systematic review and modelling analysis,” *The Lancet Respiratory Medicine*, vol. 10, no. 5, pp. 447–458, May 2022.
 - [5] E. Boers, M. Barrett, J. G. Su, A. V. Benjafield, S. Sinha, L. Kaye, H. J. Zar, V. Vuong, D. Tellez, R. Gondalia, M. B. Rice, C. M. Nunez, J. A. Wedzicha, and A. Malhotra, “Global Burden of Chronic Obstructive Pulmonary Disease Through 2050,” *JAMA Network Open*, vol. 6, no. 12, p. e2346598, Dec. 2023.
 - [6] L. Rayner, J. Sherlock, B. Creagh-Brown, J. Williams, and S. deLusignan, “The prevalence of copd in england: An ontological approach to case detection in primary care,” *Respiratory Medicine*, vol. 132, pp. 217–225, Nov. 2017.
 - [7] R. Halbert, S. Isonaka, D. George, and A. Iqbal, “Interpreting COPD Prevalence Estimates,” *Chest*, vol. 123, no. 5, pp. 1684–1692, May 2003.
 - [8] S. Scully, H. Evans, S. Hatam, S. Cook, H. Whittaker, C. Kallis, I. Farr, C. Orton, A. Sheikh, and J. Quint, “Incidence and prevalence of asthma, copd, and ild across england, wales, and south-east scotland – a harmonised approach to data analysis,” *International Journal of Population Data Science*, vol. 9, no. 5, Sep. 2024.
 - [9] J. A. Fiz, R. Jané, M. Lozano, R. Gómez, and J. Ruiz, “Detecting unilateral phrenic paralysis by acoustic respiratory analysis,” *PLoS ONE*, vol. 9, no. 4, p. e93595, Apr. 2014.
 - [10] M. Lozano-García, F. A. Paredes, C. J. Jolley, and R. Jané, “Respiratory sound intensity as a noninvasive acoustic biomarker in copd,” in *2024 46th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, Jul. 2024, pp. 1–4.
 - [11] W. Ashraf and Z. Moussavi, “Design and analysis of a contact piezo microphone for recording tracheal breathing sounds,” *Sensors*, vol. 24, no. 17, p. 5511, Aug. 2024.
 - [12] B. B. T., H. I. Hee, O. H. Teoh, K. P. Lee, S. Kapoor, D. Herremans, and J.-M. Chen, “Asthmatic versus healthy child classification based on cough and vocalised ah sounds,” *The Journal of the Acoustical Society of America*, vol. 148, no. 3, pp. EL253–EL259, Sep. 2020.
 - [13] D. Aalto, J. Malinen, and M. Vainio, “Formants,” *Oxford Research Encyclopedia of Linguistics*, Jun. 2018.
 - [14] J. H. Appleton, R. C. Perera, and O. Luening, *The development and practice of electronic music*, J. H. Appleton, R. C. Perera, and O. Luening, Eds. Englewood Cliffs, NJ: Prentice-Hall, 1975, literaturverz. S. 336 - 343.
 - [15] S. Mootassim-Billah, J. Schoentgen, M. De Bodt, N. Roper, A. Digonnet, M. Le Tensorer, G. Van Nuffelen, and D. Van Gestel, “Acoustic analysis of voluntary coughs, throat clearings, and induced reflexive coughs in a healthy population,” *Dysphagia*, vol. 38, no. 6, pp. 1467–1486, May 2023.

- [16] S. Ghrabli, M. Elgendi, and C. Menon, “Identifying unique spectral fingerprints in cough sounds for diagnosing respiratory ailments,” *Scientific Reports*, vol. 14, no. 1, Jan. 2024.
- [17] Y. Yan, L. van Bommel, F. M. Franssen, S. O. Simons, and V. Urovi, “Developing a multi-feature fusion model for exacerbation classification in asthma and copd,” *Computer Methods and Programs in Biomedicine*, vol. 268, p. 108796, Aug. 2025.
- [18] G. de Souza dos Reis, J. Baddini-Martinez², and B. Sanches Masiero¹, “Pulmonary function estimation using smartphone audio and deep learning,” *Jornal Brasileiro de Pneumologia*, p. e20250003, Jul. 2025.
- [19] R. Alsagri, A. Wilde, K. Farrahi, A. Alhomoud, and N. White, “Emerging technologies and trends in lung volume estimation: A review,” *IEEE Sensors Journal*, vol. 25, no. 13, pp. 23 552–23 563, Jul. 2025.
- [20] A. Mahboubian, M. J. Kasteleyn, E. Bašić, N. H. Chavannes, and J. J. Aardoom, “Patients’ and health care professionals’ perspectives on remote patient monitoring in chronic obstructive pulmonary disease exacerbation management: Initiating cocreation,” *Journal of Medical Internet Research*, vol. 27, p. e67666, May 2025.
- [21] S. N. Miller, S. McCabe, E. Morgan, R. Al Hourany, M. Nichols, T. Fowler, W. Smith, C. Durham, M. Pittman, M. Madisetti, R. Teufel, A. Swartz, and C. Strange, “Remote spirometry, education, and action for copd diagnosis and management in rural and medically underserved settings in south carolina: Protocol for an observational feasibility study,” *Journal of Evaluation in Clinical Practice*, vol. 31, no. 1, Feb. 2025.
- [22] M. Galvosas, J. C. Gabaldón-Figueira, E. M. Keen, V. Orrillo, I. Blavia, J. Chaccour, P. M. Small, G. Giménez, M. Rudd, S. Grandjean Lapierre, and C. Chaccour, “Performance evaluation of the smartphone-based ai cough monitoring app - hyfe cough tracker against solicited respiratory sounds,” *F1000Research*, vol. 11, p. 730, Jun. 2023.
- [23] S. G. Jeong and S. E. Kim, “Patient domain supervised contrastive learning for lung sound classification using mobile phone,” 2025.
- [24] A. Systems, “Eupnoos respiratory app for lung function tests,” Website, Oct. 2025. [Online]. Available: <https://www.atta.systems/case-studies/eupnoos#:~:text=Eupnoosisaninnovativerespiratory,tri,als,anduserexperiencedesign>.
- [25] L. Welch, M. Shaban, C. Sheppard, A. Gratiot, D. Kodak, and S. Siddiqui, “S134 eupnoos: advancing early diagnosis of respiratory diseases with smartphone-based audio phenotyping,” *Thorax*, vol. 78, no. Suppl 4, pp. A97–A98, 2023. [Online]. Available: https://thorax.bmj.com/content/78/Suppl_4/A97.2
- [26] Y. Kim, Y. Hyon, S. Lee, S.-D. Woo, T. Ha, and C. Chung, “The coming era of a new auscultation system for analyzing respiratory sounds,” *BMC Pulmonary Medicine*, vol. 22, no. 1, Mar. 2022.
- [27] K. Safavi, S. C. Mathews, D. W. Bates, E. R. Dorsey, and A. B. Cohen, “Top-funded digital health companies and their impact on high-burden, high-cost conditions,” *Health Affairs*, vol. 38, no. 1, pp. 115–123, Jan. 2019.

- [28] A. Khanzada, S. Hegde, S. Sreeram, G. Bower, W. Wang, R. P. Mediratta, K. D. Meister, and A. Rameau, "Challenges and opportunities in deploying covid-19 cough ai systems," *Journal of Voice*, vol. 35, no. 6, pp. 811–812, Nov. 2021.
- [29] F. Busquet, F. Efthymiou, and C. Hildebrand, "Voice analytics in the wild: Validity and predictive accuracy of common audio-recording devices," *Behavior Research Methods*, vol. 56, no. 3, pp. 2114–2134, May 2023.
- [30] J. G. Svec and S. Granqvist, "Guidelines for Selecting Microphones for Human Voice Production Research," *American Journal of Speech-Language Pathology*, vol. 19, no. 4, pp. 356–368, Nov. 2010.
- [31] D. Pines, "RODE NT1 Review: The Condenser Mic for You?" <https://mastering.com/rode-nt1-review/>, May 2018, accessed on 21-01-2026. [Online]. Available: <https://mastering.com/rode-nt1-review/>
- [32] W3C, "Web Audio API enables the creation and manipulation of music and sounds, directly in the browser," <https://www.w3.org/press-releases/2021/webaudio/>, Jun. 2021, accessed on 21-01-2026. [Online]. Available: <https://www.w3.org/press-releases/2021/webaudio/>
- [33] MDN, "Web Audio API," Jan. 2026, accessed on 21-01-2026. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API
- [34] D. Moerkerke, "PWA Audio recording," <https://whatpwacando.today/audio-recording>, Jan. 2026, accessed on 21-01-2026. [Online]. Available: <https://whatpwacando.today/audio-recording>
- [35] N. Jillings, Y. Wang, R. Stables, and J. Reiss, "Intelligent audio plugin framework for the Web Audio API," in *Intelligent audio plugin framework for the Web Audio API*, 08 2017.
- [36] B. Matuszewski and O. Rottier, "The Web Audio API as a Standardized Interface Beyond Web Browsers," *Journal of the Audio Engineering Society*, vol. 71, no. 11, pp. 790–801, Nov. 2023.
- [37] P. Adenot and H. Choi, "Web Audio API 1.1," <https://www.w3.org/TR/webaudio-1.1/>, Nov. 2024, accessed on 21-01-2026. [Online]. Available: <https://www.w3.org/TR/webaudio-1.1/>
- [38] J. Berkovitz and O. Theroux, "Web Audio Processing: Use Cases and Requirements," <https://www.w3.org/TR/webaudio-usecases/>, Jan. 2013, accessed on 22-01-2026. [Online]. Available: <https://www.w3.org/TR/webaudio-usecases/>
- [39] S. T. Yousif and B. M. Mahmmoud, "Speech Enhancement Algorithms: A Systematic Literature Review," *Algorithms*, vol. 18, no. 5, p. 272, May 2025.
- [40] W. Zhang, J. Benesty, and J. Chen, "Single-channel noise reduction via semi-orthogonal transformations and reduced-rank filtering," *Speech Communication*, vol. 78, pp. 73–83, Apr. 2016.
- [41] W. A. Jassim, R. Paramesran, and M. S. Zilany, "Enhancing noisy speech signals using orthogonal moments," *IET Signal Processing*, vol. 8, no. 8, pp. 891–905, Oct. 2014.
- [42] B. Widrow, J. Glover, J. McCool, J. Kaunitz, C. Williams, R. Hearn, J. Zeidler, J. Eugene Dong, and R. Goodlin, "Adaptive noise cancelling: Principles and applications," *Proceedings of the IEEE*, vol. 63, no. 12, pp. 1692–1716, 1975.
- [43] N. Upadhyay and R. K. Jaiswal, "Single Channel Speech Enhancement: Using Wiener Filtering with Recursive Noise Estimation," *Procedia Computer Science*, vol. 84, pp. 22–30, 2016.

- [44] Anvekar, Rohan V. and Hosalli, Rohit M. and Shindolkar, Samruddhi V. and Tadasalur, Ujwala V. and Manage, Prabhakar S, “White noise Denoising of audio signal using Auto encoder, Band pass Filter and Wiener Filter,” in *2025 6th International Conference for Emerging Technology (INCET)*. IEEE, May 2025, pp. 1–4.
- [45] A. Hashmi, “Perceptual evaluation of speech quality for inexpensive recording equipment,” *Acoustics*, vol. 3, no. 1, pp. 200–211, Mar. 2021.
- [46] A. N. SLOSS, D. SYMES, and C. WRIGHT, *DIGITAL SIGNAL PROCESSING*. Elsevier, 2004, pp. 258–314.
- [47] Z. Ling, Y. Zeng, and P. He, “Research on the Principle, Application and Optimization of Spectral Subtraction Audio Noise Reduction Technology,” in *2025 5th International Conference on Electronic Information Engineering and Computer Science (EIECS)*. IEEE, Sep. 2025, pp. 1084–1088.
- [48] S. Boll, “Suppression of acoustic noise in speech using spectral subtraction,” *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 27, no. 2, pp. 113–120, Apr. 1979.
- [49] M. T. Sadiq, N. Shabbir, and W. J. Kulesza, “Spectral Subtraction for Speech Enhancement in Modulation Domain,” *International Journal of Computer Science Issues (IJCSI)*, vol. 10, no. 4, pp. 282–285, 07 2013, copyright - Copyright International Journal of Computer Science Issues (IJCSI) Jul 2013; Last updated - 2023-11-20. [Online]. Available: <https://www.proquest.com/scholarly-journals/spectral-subtraction-speech-enhancement/docview/1470806268/se-2>
- [50] L.-y. Sui, X.-w. Zhang, J.-j. Huang, and B. Zhou, “An improved spectral subtraction speech enhancement algorithm under non-stationary noise,” in *2011 International Conference on Wireless Communications and Signal Processing (WCSP)*. IEEE, Nov. 2011, pp. 1–5.
- [51] J. Chen, J. Benesty, Y. Huang, and S. Doclo, “New insights into the noise reduction Wiener filter,” *IEEE Transactions on Audio, Speech and Language Processing*, vol. 14, no. 4, pp. 1218–1234, Jul. 2006.
- [52] J. Zhu, W. Cai, M. Zhang, and Y. Yang, “Self-supervised denoising model based on deep audio prior using single noisy marine mammal sound sample,” *Applied Intelligence*, vol. 53, no. 21, pp. 25 697–25 714, Aug. 2023.
- [53] Z. Chen, “Implementation of Music Noise Removal Technology based on Adaptive Filtering Algorithm,” in *2025 4th International Conference on Distributed Computing and Electrical Circuits and Electronics (ICDCECE)*. IEEE, Apr. 2025, pp. 1–6.
- [54] Y. Bai, M. Fan, Y. Li, and C. Xie, “Privacy Risk Assessment of Training Data in Machine Learning,” in *ICC 2022 - IEEE International Conference on Communications*. IEEE, May 2022, pp. 1015–1015.
- [55] S. Shahnawazuddin, C. Singh, H. K. Kathania, W. Ahmad, and G. Pradhan, “An experimental study on the significance of variable frame-length and overlap in the context of children’s speech recognition,” *Circuits, Systems, and Signal Processing*, vol. 37, no. 12, pp. 5540–5553, May 2018.
- [56] K. Khaldi, A. Boudraa, and M. Turki, “Voiced/unvoiced speech classification-based adaptive filtering of decomposed empirical modes for speech enhancement,” *IET Signal Processing*, vol. 10, no. 1, pp. 69–80, Feb. 2016.

- [57] R. K. Parida, G. Singh, and A. N. Pal, “Optimizing React Components For Improving The Performance Of MERN Stack Application,” in *2025 Seventh International Conference on Computational Intelligence and Communication Technologies (CCICT)*. IEEE, Apr. 2025, pp. 55–61.
- [58] S. Wang, G. Li, X. Yao, Y. Zeng, L. Pang, and L. Zhang, “A Distributed Storage and Access Approach for Massive Remote Sensing Data in MongoDB,” *ISPRS International Journal of Geo-Information*, vol. 8, no. 12, p. 533, Nov. 2019.
- [59] L. Liang, L. Zhu, W. Shang, D. Feng, and Z. Xiao, “Express supervision system based on NodeJS and MongoDB,” in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*. IEEE, May 2017, pp. 607–612.
- [60] O. Chaplia, H. Klym, M. Konuhova, and A. I. Popov, “Enhancing REST API Handlers Organization for Node.js Microservices,” *SN Computer Science*, vol. 6, no. 7, Aug. 2025.
- [61] M. Obczovsky, A. Bernsteiner, C. Haagen-Schützenhöfer, and T. Schubatzky, “Systematizing decisions in design-based research: From theory to design,” *Science Education*, vol. 109, no. 2, pp. 523–536, Oct. 2024.
- [62] J. R. Venable, J. Pries-Heje, and R. L. Baskerville, “Choosing a design science research methodology,” in *ACIS*, 2017. [Online]. Available: <https://api.semanticscholar.org/CorpusID:115194465>
- [63] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, Dec. 2007.
- [64] R. Bevans, “A Guide to Experimental Design - 5 Steps and Examples,” Dec. 2022, accessed on 25-01-2026. [Online]. Available: <https://www.scribbr.co.uk/research-methods/guide-to-experimental-design/>
- [65] S. Komai, H. Nakanishi, and H. Saidi, “Guidelines for selecting agile development method in system requirements definition,” in *2017 7th IEEE International Conference on Control System, Computing and Engineering (ICCSCE)*. IEEE, Nov. 2017, pp. 49–53.
- [66] P. Beynon-Davies, C. Carne, H. Mackay, and D. Tudhope, “Rapid application development (rad): an empirical review,” *European Journal of Information Systems*, vol. 8, no. 3, pp. 211–223, Sep. 1999.
- [67] A. Butterfield, *A Dictionary of Computer Science*, 7th ed., ser. Oxford Quick Reference Ser., G. E. Ngondi and A. Kerr, Eds. Oxford: Oxford University Press, Incorporated, 2016, description based on publisher supplied metadata and other sources.
- [68] MDN, “MediaTrackConstraints: channelCount property,” Website, May 2024, accessed on 05-04-2026. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/MediaTrackConstraints/channelCount>