Transilvania University of Brasov

Faculty of Mathematics and Computer Science

~ Computer Science ~

Course: Human-computer interaction (HCI)

Piano

~Evolution from a simple IoT project to a true instrument ~

Team: Trei Muschetari

Authors: Stancu Nicol-Alexia, Vlaicu Andrei-Danielo, Vîlcu Alessandra-Marina, Neagu Roxana

BRAȘOV

~2025 - 2026~

Contents

1. What piano really is: system overview

This chapter gives an introduction to the Piano system, explaining what it does, who it is designed for, and the main ideas behind its design. It shows how the system uses multimodal interaction and accessibility features to make learning music easier for everyone. You will also see how Piano can be used not only as a tool for practicing music, but also as an educational platform for exploring interactive IoT systems in a hands-on way.

1.1 Motivation and core principles

Piano was designed not only as a technical demonstration (IoT + AR + cloud AI), but as an accessible learning system for users who may struggle with traditional music learning interfaces. The core design principle of the system is summarized by the following statement:

"We create interfaces for the user, not users for the interface."

The system aims to bridge the gap between interactive technology and inclusive music education. It demonstrates how a simple instrument can evolve into a complex, multimodal platform that supports learning, exploration, and accessibility.

1.2 Target user groups

The system targets three main user groups:

- ➢ Older adults, who benefit from slower interaction flows and clear, step-by-step guidance.
- ➢ People with disabilities (motor, visual, hearing, or cognitive), who require alternative feedback channels and more tolerant interaction mechanisms.
- ➢ Educational users (students and/or beginners):
  - Educational music users (students and beginners), for whom the system provides a guided environment for introducing fundamental music concepts through immediate and multimodal feedback.
  - Educational technical users (students and early practitioners), who can use Piano as an entry-level platform for understanding IoT-based interactive systems. The project's evolution from a simple instrument to a more complex, multi-component system allows learners to explore increasing technical complexity while remaining accessible and engaging.

1.3 Multimodal interaction and feedback

Instead of relying on a single output modality (e.g., sound only), Piano intentionally uses a multimodal interaction model. The system combines physical input (piano keys) with audio, visual, and textual feedback delivered through AR, a web interface, and an LCD display. This approach improves usability and significantly reduces learning barriers.

Many music learning tools assume that users can hear and correctly interpret musical feedback. Piano provides redundant feedback across multiple channels:

- ➢ Audio feedback (buzzer), allowing immediate auditory confirmation of the played note.
- ➢ Visual feedback (AR spheres and sheet-music-style UI), indicating which key should be pressed and whether the action was correct.
- ➢ Textual feedback (LCD and UI labels), displaying note names and the recognized melody.
- ➢ Tactile feedback (physical buttons), offering strong physical cues that support users with limited vision or reduced fine motor control.

This redundancy is essential because different users may rely on different sensory channels:

- ➢ Users with reduced hearing can depend more on visual and textual feedback.
- ➢ Users with reduced vision can rely on audio cues and simplified interaction flows.
- ➢ Users with cognitive or memory limitations benefit from step-by-step guidance and immediate confirmation.

1.4 Design goals and intended use

The Piano system was designed with the following goals:

- ➢ to provide an accessible and inclusive environment for assisted music learning, accommodating users with varying levels of ability, age, and prior experience;
- ➢ to support multimodal interaction through redundant sensory feedback, including audio, visual, textual, and tactile channels;
- ➢ to maintain a low entry barrier for beginners while allowing system scalability and increasing technical complexity for advanced exploration;

- ➢ to serve as both a learning instrument and an educational platform, introducing users not only to fundamental musical concepts but also to interactive IoT-based systems, AR interfaces, and AI-driven feedback;

- ➢ to offer a guided and structured learning experience, with step-by-step instructions, immediate feedback, and adaptive support for cognitive and memory limitations;

- ➢ to demonstrate how a simple system can evolve into a complex, multi-component interactive platform while remaining user-friendly and engaging;

- ➢ to function as a research and demonstration tool for future studies in human-computer interaction, accessibility, and educational technology.

The system is intended for educational and exploratory use and is not meant to replace professional musical instruments. Instead, it provides a safe, flexible, and engaging environment where learners can experiment, practice, and gradually explore more complex technical and musical concepts.

2. Hardware evolution

This chapter walks you through the evolution of the Piano hardware, starting from the original Arduino-based prototype and moving to the upgraded ESP32 version. It explains the role of each component, how the system operates step by step, and the improvements introduced by the hardware update.

Additionally, it demonstrates how these changes enhance the system's capabilities while keeping the user experience simple and intuitive.

2.1 Arduino-based version: system overview

In the Arduino-based version, the Arduino Uno acts as the central controller of the Piano system. Its main responsibility is to continuously read the state of the keys, determine which musical note has been pressed, and provide immediate feedback through both sound, using a piezoelectric buzzer, and text, displayed on a 16x2 LCD. This initial setup allowed the prototype to demonstrate the core functionality of the piano in a straightforward and effective manner, as shown in Fig. 1.

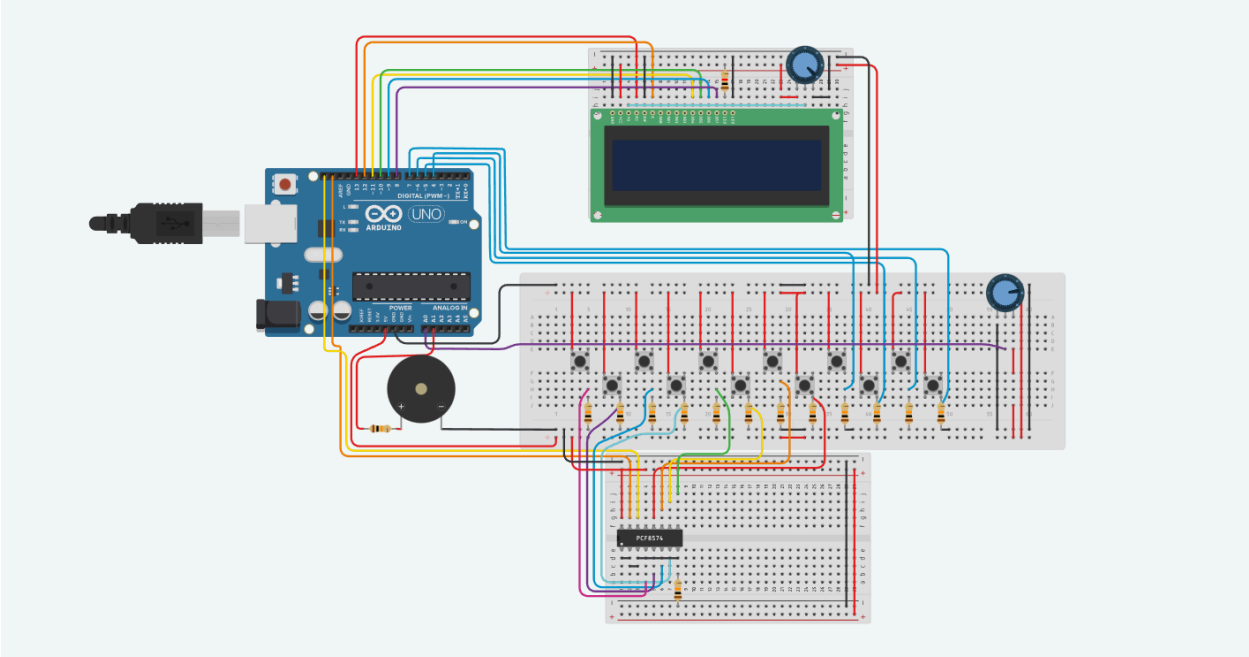


Fig 1: Hardware scheme in TinkerCad with Arduino Uno

2.1.1 Components overview

The Arduino-based piano uses a combination of standard electronic components:

1) Arduino Uno

Arduino Uno is a microcontroller board based on the ATmega328P (MCU). Released in 2010 by Arduino.cc, it features 14 digital and 6 analog input/output pins that can be programmed using the Arduino IDE via a Type-B USB cable. Its open-source nature, combined with strong community support, makes it a versatile choice for both beginners and advanced developers.

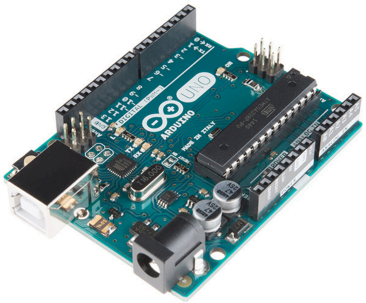
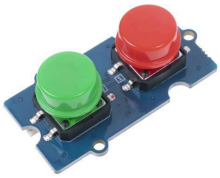


Fig 2: Arduino Uno



2) Tactile buttons

Tactile buttons serve as the piano keys. These momentary switches close the circuit when pressed and send a digital HIGH or LOW signal to the Arduino, allowing it to detect key presses accurately.

Fig 3: Tactile buttons

3) Sound generation module: piezoelectric buzzer

Piezoelectric buzzer is used to generate sound corresponding to each pressed note. The buzzer contains a piezo element that oscillates when voltage is applied, producing audible tones. It is connected to a PWM-capable Arduino pin to allow frequency control.

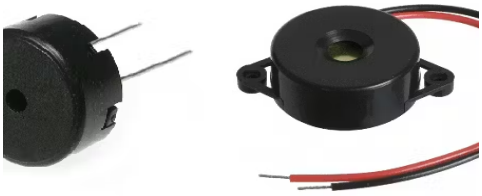
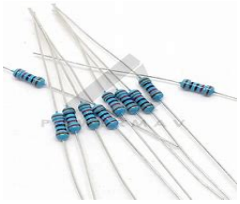


Fig 4: Piezoelectric buzzer



4) Resistors

Resistors are included to limit current flow and protect other components such as LEDs, buzzers, or the LCD from excessive current.

Fig 5: Resistors

5) Breadboard and connecting wires

Breadboards and jumper wires are used to assemble the circuit without soldering, enabling easy prototyping and testing. Jumper wires of various colors connect components such as buttons, resistors, and the Arduino board itself.

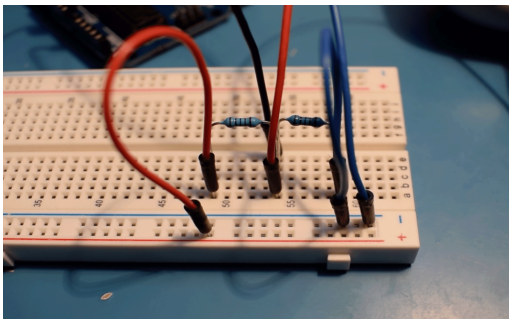
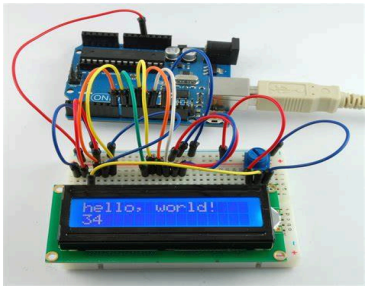


Fig 6: Breadboard



6) LCD display

LCD display (16x2**)** provides textual feedback, showing the name of the pressed note and status messages like “No key pressed” or “Note:E4, Hz: 214”. Its contrast can be adjusted via a dedicated potentiometer.

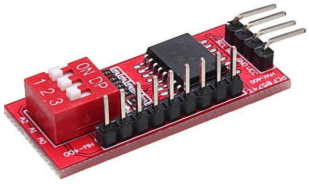
Fig 7: LCD display

7) Potentiometer

Potentiometers are variable resistors that allow control of certain electrical parameters. In this setup, they are used to adjust the LCD contrast and, optionally, the output sound intensity.

Fig 8: Potentiometer

8) Expander



PCF8574 I²C expander extends the number of digital input pins available to the Arduino using only two communication lines: SDA (Serial Data) and SCL (Serial Clock). This allows multiple keys to be connected without using all of the Arduino’s GPIO pins.

Fig 9: Expander PCF8574

2.1.2 System operation

The Piano has twelve tactile buttons representing the notes, eight of which are connected through the PCF8574 I²C expander, while the remaining four connect directly to Arduino digital input pins. This configuration optimizes the use of GPIO pins while supporting all keys and the LCD display. The Arduino continuously reads the I²C expander, receiving an 8-bit value where each bit corresponds to a key, allowing the controller to determine which keys are pressed at any given moment. Mechanical buttons can generate multiple rapid signals for a single press, a phenomenon known as “bouncing.” To prevent false readings, a debounce strategy is applied, ensuring that each press is registered only once.

Once a key press is confirmed, the Arduino maps the pressed key to its corresponding musical note and frequency using a predefined lookup table. The piezo buzzer immediately generates the correct tone, while the 16x2 LCD updates to display the note name and a short status message. Potentiometers allow users to adjust the LCD contrast and, if included, the sound intensity, enhancing visibility and auditory experience. This workflow combines tactile, auditory, and visual feedback, making the system intuitive and suitable as a learning tool.

2.2 ESP32 variant: hardware update

In the upgraded version, the core components (twelve tactile buttons, the PCF8574 I²C expander, piezo buzzer, 16x2 LCD, potentiometers, resistors, breadboard, and jumper wires) remain unchanged, ensuring that the interaction remains familiar to the user (Fig. 10).

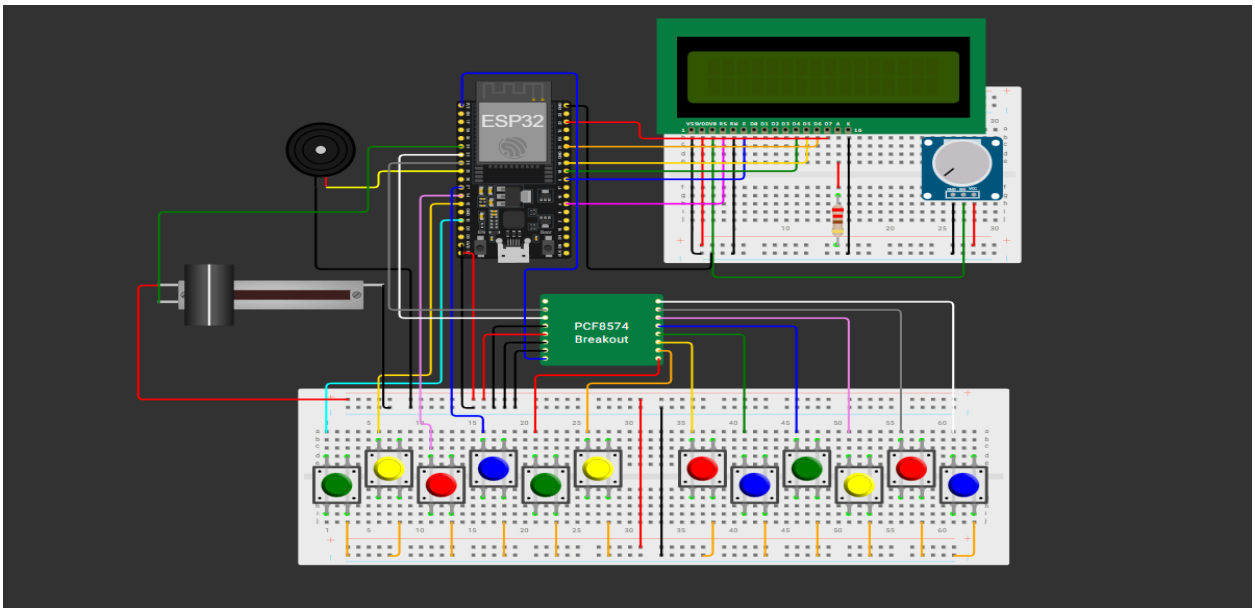


Fig 10: Hardware scheme with ESP32

The main difference is the replacement of the Arduino Uno with an ESP32 development board, which performs the same fundamental tasks of reading key presses, mapping notes, and controlling outputs, but introduces several advantages.

The ESP32 features built-in Wi-Fi, enabling the Piano to communicate directly with cloud services, such as sending note sequences to Azure for melody recognition, without additional hardware. It also offers greater processing power and memory, allowing more complex functionality, buffering of note sequences, and support for multiple modes simultaneously. These improvements simplify future expansions, including networking, real-time streaming, and additional IoT features. Despite these technical enhancements, user interaction remains immediate and intuitive, with each key press producing consistent sound and textual feedback.

2.3 Summary

The transition from Arduino Uno to ESP32 demonstrates how the Piano system can evolve from a simple, locally-operated prototype to a fully connected IoT device. Core design principles such as tactile input, multimodal feedback, and accessibility are preserved, while technical performance, cloud connectivity, and system scalability are significantly enhanced. This ensures that the Piano continues to function both as a practical learning tool for music and as an educational platform for exploring interactive technologies and IoT-based systems.

3. Software evolution

3.1 Arduino firmware (initial prototype)

The firmware for the first hardware version of the Piano was built around the Arduino Uno, which acted as the main controller. In this initial stage, the Arduino was responsible for reading the twelve piano keys, eight through the PCF8574 I²C expander and four directly from digital pins, mapping each key press to a musical note, displaying the note name on a 16x2 LCD, and playing the corresponding tone on a piezo buzzer. A potentiometer was used to apply a small frequency offset, allowing minor adjustments to the sound output. This setup enabled a simple yet functional interaction loop, providing immediate auditory and visual feedback for each key press.

The Arduino variant was implemented in Arduino-style C/C++, using the Arduino IDE, which provides a complete toolchain, serial monitor for debugging, and simplified uploading of the firmware. Serial communication was initialized using Serial.begin(9600) to allow diagnostic messages, including generated frequencies, to be printed during runtime.

```
#include <Wire.h>
#include <LiquidCrystal.h>
#include <PCF8574.h>
LiquidCrystal lcd(13, 12, 11, 10, 9, 8);
#define EXP_ADDR 0x20
PCF8574 expander(EXP_ADDR);
const uint8_t directPins[4] = { 4, 5, 6, 7 };
#define BUZZER_PIN A1
#define POT_PIN A0

const uint16_t noteFreq[12] = {
262, 277, 294, 311, 330, 349,
370, 392, 415, 440, 466, 494
};

const char* noteName[12] = {
"C4","C#4","D4","D#4","E4","F4",
"F#4","G4","G#4","A4","A#4","B4"
};

void setup() {
Serial.begin(9600);
Wire.begin();
expander.begin();
```

```
void loop() {
bool cur0 = expander.digitalRead(P0);
if (cur0 == HIGH) {playNote(0); expander.digitalWrite(P0, LOW);}
bool cur1 = expander.digitalRead(P1);
if (cur1 == HIGH) {playNote(1); expander.digitalWrite(P0, LOW);}
bool cur2 = expander.digitalRead(P2);
if (cur2 == HIGH) {playNote(2); expander.digitalWrite(P0, LOW);}
bool cur3 = expander.digitalRead(P3);
if (cur3 == HIGH) {playNote(3); expander.digitalWrite(P0, LOW);}
bool cur4 = expander.digitalRead(P4);
if (cur4 == HIGH) {playNote(4); expander.digitalWrite(P0, LOW);}
bool cur5 = expander.digitalRead(P5);
if (cur5 == HIGH) {playNote(5); expander.digitalWrite(P0, LOW);}
bool cur6 = expander.digitalRead(P6);
if (cur6 == HIGH) {playNote(6); expander.digitalWrite(P0, LOW);}
bool cur7 = expander.digitalRead(P7);
if (cur7 == HIGH) {playNote(7); expander.digitalWrite(P0, LOW);}
static bool prevD4 = HIGH;
bool curD4 = digitalRead(4);
if (prevD4 == HIGH && curD4 == LOW) playNote(8);
prevD4 = curD4;
static bool prevD5 = HIGH;
bool curD5 = digitalRead(5);
if (prevD5 == HIGH && curD5 == LOW) playNote(9);
prevD5 = curD5;
```

```
expander.pinMode(P0, INPUT);
expander.pinMode(P1, INPUT);
expander.pinMode(P2, INPUT);
expander.pinMode(P3, INPUT);
expander.pinMode(P4, INPUT);
expander.pinMode(P5, INPUT);
expander.pinMode(P6, INPUT);
expander.pinMode(P7, INPUT);
pinMode(4, INPUT_PULLUP);
pinMode(5, INPUT_PULLUP);
pinMode(6, INPUT_PULLUP);
pinMode(7, INPUT_PULLUP);
pinMode(BUZZER_PIN, OUTPUT);
lcd.begin(16,2);
lcd.print("Arduino Piano");}
```

```
static bool prevD6 = HIGH;
bool curD6 = digitalRead(6);
if (prevD6 == HIGH && curD6 == LOW) playNote(10);
prevD6 = curD6;
static bool prevD7 = HIGH;
bool curD7 = digitalRead(7);
if (prevD7 == HIGH && curD7 == LOW) playNote(11);
prevD7 = curD7;
delay(50);
}

void playNote(uint8_t idx) {
if (idx > 11) return;
int potValue = analogRead(POT_PIN);
int freqOffset = map(potValue, 0, 1023, 0, 200);
int f = noteFreq[idx] + freqOffset;
lcd.clear();
lcd.setCursor(0,0);
lcd.print("Nota: ");
lcd.print(noteName[idx]);
lcd.setCursor(0,1);
lcd.print("+");
lcd.print(freqOffset);
lcd.print("Hz");
Serial.print("Frecv: ");
Serial.print(f);
Serial.println(" Hz");
tone(BUZZER_PIN, f);
delay(300);
noTone(BUZZER_PIN);
}
```

Arduino code (first prototype version)

The software architecture of the initial Arduino-based prototype is organized around a simple, linear execution model typical for Arduino applications. The system can be conceptually divided into three main areas: inputs, application logic, and outputs, forming a deterministic processing flow from user interaction to system feedback.

The input layer consists of twelve piano keys and a potentiometer. Eight keys are read via the PCF8574 I²C expander, while the remaining four keys are connected directly to Arduino digital pins. These inputs are continuously scanned inside the main loop() function, ensuring constant responsiveness to user actions.

When a key press is detected, the application logic maps the pressed key to a corresponding musical note. The final output frequency is computed by combining the base note frequency with a small offset derived from the potentiometer value. This processing step is encapsulated in the playNote() function.

The output layer provides immediate feedback to the user. The buzzer generates the

sound using the built-in tone() function, while the LCD displays the note name and frequency offset through the LiquidCrystal library. The single-threaded, sequential execution model ensures immediate and deterministic feedback for each interaction.

From a human–computer interaction perspective, this architecture establishes a clear and direct mapping between user action (pressing a piano key) and system response (auditory and visual confirmation), resulting in low latency and intuitive interaction.

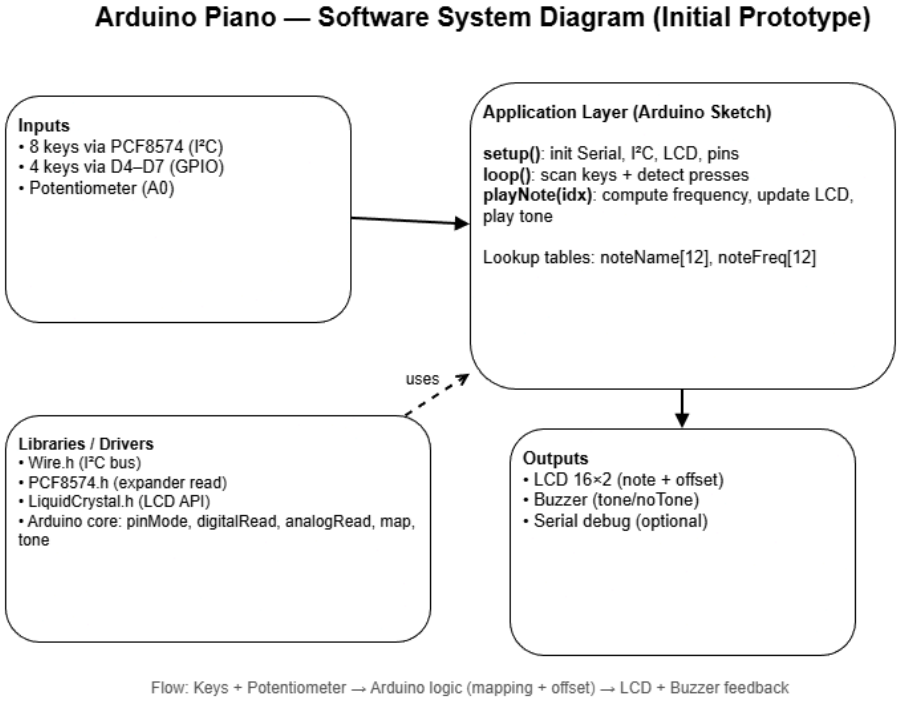


Fig 11

3.2 Software technologies and libraries

To maintain readability, reliability, and modularity, several well-established Arduino libraries were used. The firmware abstracts low-level hardware details, enabling straightforward interaction with components such as the I²C expander, LCD, and buzzer.

Wire.h is used for I²C communication with the PCF8574 expander. Including this library and calling Wire.begin() initializes the Arduino as an I²C master device. This allows reading the state of eight keys using only two wires, SDA and SCL, without manually implementing start and stop conditions or acknowledgements, simplifying the code and improving reliability.

PCF8574.h manages the digital input expansion. The PCF8574 provides additional I/O pins, enabling the Arduino to read multiple button states without consuming a large number of onboard pins. In the code, the expander is instantiated with PCF8574 expander(EXP_ADDR) and initialized with expander.begin(). Pins are configured as inputs via expander.pinMode(P0..P7, INPUT) and read in the main loop using expander.digitalRead(P0..P7). This design allowed keys zero through seven to be handled by the expander while keeping four keys directly connected to Arduino pins four through seven in INPUT_PULLUP mode, reducing dependency on the expander for every input and simplifying the circuit.

LiquidCrystal.h handles the LCD, providing textual feedback for the user. The display shows the note being played and the frequency offset derived from the potentiometer, supporting usability by providing visual confirmation in addition to sound. In the code, the LCD is defined with LiquidCrystal lcd(13, 12, 11, 10, 9, 8) and initialized with lcd.begin(16,2). Functions such as lcd.setCursor(), lcd.print(), and lcd.clear() update the display whenever a key is pressed, typically inside the playNote() function.

Built-in Arduino functions such as tone() and noTone() are used for sound generation. tone(BUZZER_PIN, f) produces a square-wave signal at the desired frequency on the buzzer pin, while noTone(BUZZER_PIN) stops the sound. This approach is simple, efficient, and sufficient for the prototype, allowing each note to be played immediately upon key press.

3.3 ESP32 implementation

Following the Arduino prototype stage, the firmware was migrated to an ESP32 project using the ESP-IDF framework in Visual Studio Code. The main goal of this transition was to move from a single-file sketch to a modular, driver-based architecture. Each hardware component, including buttons, buzzer, LCD, and potentiometer, is encapsulated in its own module, enabling clearer code organization and maintainability. The application leverages FreeRTOS tasks for concurrent operation, with separate tasks responsible for button scanning, potentiometer sampling, buzzer output, and LCD updates. Additionally, the system includes Wi-Fi Station mode and a Server-Sent Events (SSE) server that streams live note events to an Angular web frontend, providing real-time interactive feedback.

The hardware design remains largely unchanged from the Arduino stage. The twelve push buttons, piezo buzzer, LCD 16x2, potentiometer, and PCF8574 I²C expander continue to serve the same functions. The main difference is the ESP32 controller, which allows multitasking, native networking, and real-time web integration. Unlike the Arduino firmware, which relied on a single loop structure, the ESP32 firmware uses FreeRTOS tasks to handle concurrent operations, improving responsiveness and scalability for future cloud or augmented reality features.

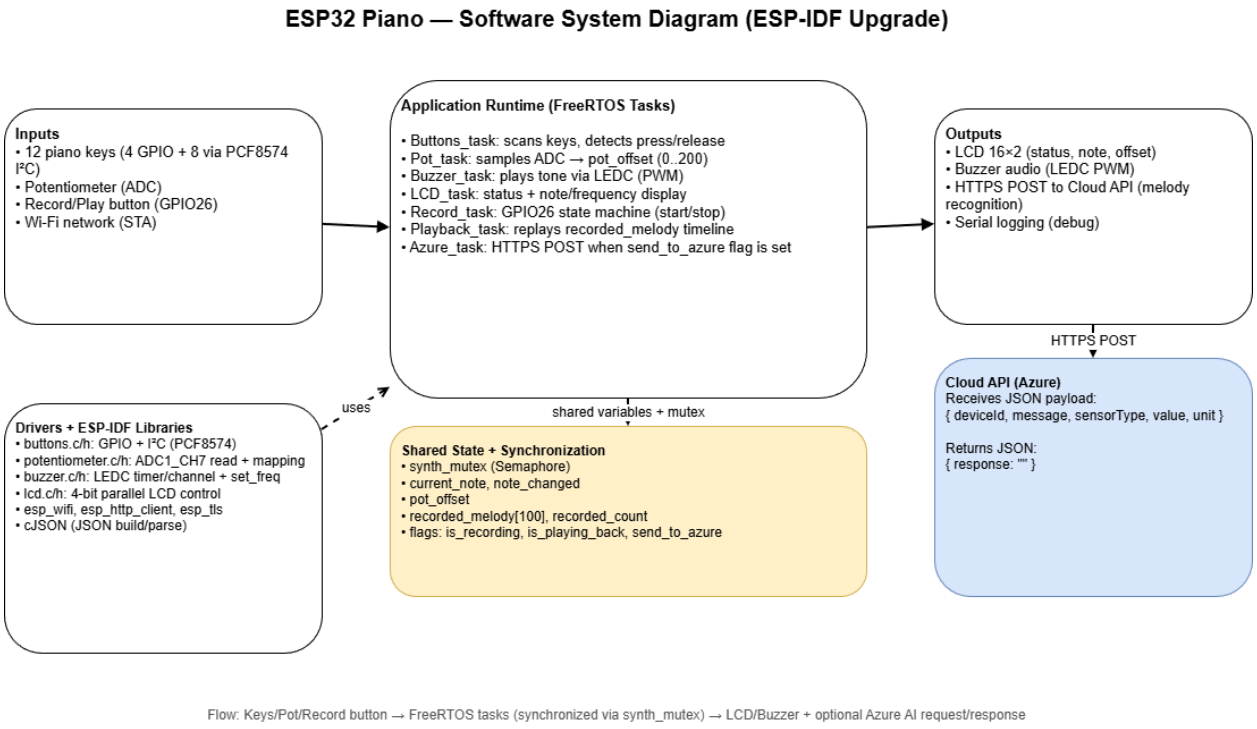


Fig 12

Figure 12 illustrates the upgraded ESP32 software architecture implemented using ESP-IDF. The diagram highlights the transition from a linear execution model to a concurrent, task-based system built on FreeRTOS. User inputs, including the twelve piano keys read via GPIO and the PCF8574 expander, the potentiometer, control buttons, and network connectivity, are handled by dedicated tasks that operate independently of one another.

Button press events are generated by the button scanning task and propagated to other system components through a shared state protected by synchronization mechanisms such as mutexes and control flags. The buzzer and LCD tasks react to these events in parallel, producing immediate audio and visual feedback without blocking other system operations. The potentiometer task continuously samples the analog input and updates control parameters used by the sound generation logic.

When a melody recording session is completed, a control flag activates the cloud communication task, which packages the recorded note sequence into a JSON payload and transmits it to the Azure API via HTTPS. The response received from the cloud service, typically a recognized melody name, is then displayed on the LCD and streamed to connected web clients using Server-Sent Events.

This diagram justifies the architectural decision to adopt a multitasking approach. Network communication and cloud-based processing are inherently slow and resource-intensive operations; isolating them in dedicated tasks prevents them from degrading the responsiveness of the user interface. From a human–computer interaction perspective, the ESP32 architecture preserves low-latency auditory and visual feedback while enabling advanced features such as recording, playback, and AI-assisted melody recognition.

3.3.1 Firmware modules

The ESP32 firmware is organized into driver-based modules, each handling a specific subsystem. The main drivers include:

Module / Driver	Files	Main responsibility	Notes
Buttons driver	buttons.c, buttons.h	Initializes and reads piano keys from GPIO and/or PCF8574	Provides clean key press events to the rest of the system
Buzzer driver	buzzer.c, buzzer.h	Generates tones and controls buzzer output	Plays frequency for each note, optionally modified by potentiometer offset
LCD driver	lcd.c, lcd.h	Controls the 16x2 LCD	Displays note name, status, and recognized melody name.
Potentiometer driver	potentiometer.c, potentiometer.h	Reads analog value (ADC) and converts it into an offset	Used to adjust pitch offset or similar control parameter.
Main application	main.c	Orchestrates system logic, starts tasks, handles Wi-Fi and SSE	Acts as the integration layer between drivers and external interfaces.

ESP32 firmware modules (driver-based architecture)

3.3.2 FreeRTOS tasks

The runtime behavior of the system is managed using FreeRTOS tasks, which operate concurrently to ensure smooth and responsive performance. Each task has a distinct role:

Task	What it reads	What it produces	Why it exists
Button scan task	GPIO keys + PCF8574 over I²C	Note events (e.g., “C4 pressed”)	Ensures stable key scanning without blocking other modules.
Potentiometer task	ADC value	Pitch offset / control parameter	Keeps analog reading independent and smooth.
Buzzer output task	Note + offset	PWM/tone output on buzzer pin	Produces real-time audio feedback with minimal latency.
LCD update task	Current note / status / melody name	LCD text updates	Provides constant user feedback for the user
SSE server / Network task	Client connections	Live note stream to browser clients	Enables real-time integration with the Angular web tutor.

FreeRTOS tasks (runtime responsibilities)

3.3.3 Hardware subsystem (ESP32)

The ESP32 implementation maintains the same core components, with pin assignments optimized for the ESP32 architecture. The system continues to read all twelve keys, generate tones, and display feedback in real-time.

Component	ESP32 Pin / Interface	Function
Buzzer	GPIO25	PWM output for tone generation
Potentiometer	GPIO35 (ADC1_CH7)	Analog read for pitch offset
Direct GPIO buttons (4 keys)	GPIO12, GPIO13, GPIO14, GPIO27	Digital inputs for remaining keys
PCF8574 expander	SDA GPIO32, SCL GPIO33	Reads 8 keys over I²C
PCF8574 address	0x24	Must match I²C wiring

LCD 16x2 (4-bit mode)	RS GPIO4, E GPIO5, D4 GPIO18, D5 GPIO19, D6 GPIO21, D7 GPIO22	Controls LCD display

ESP32 pin mapping (current hardware build)

The ESP32 firmware combines modularity, multitasking, and networking, allowing the Piano to maintain its immediate feedback and intuitive interface while enabling cloud features and web integration. Tasks handle buttons, potentiometer, buzzer, and LCD updates independently, ensuring reliable operation even when multiple keys are pressed simultaneously. The potentiometer reading is mapped to a pitch offset, providing dynamic control over tone generation, while the LCD continuously updates with note and frequency information. The SSE server enables live note streaming, connecting the physical piano to the Angular-based tutor interface in real time.

3.3.4 Technology Stack Comparison

Category	Arduino implementation	ESP32 implementation
IDE / Tooling	Arduino IDE	Visual Studio Code + ESP-IDF extension
Runtime model	Single loop (setup() + loop())	FreeRTOS tasks (concurrent modules)
Networking	Not native (usually Serial / extra modules)	Native Wi-Fi STA mode
Code structure	One main sketch (monolithic)	Driver-based modules (buttons, buzzer, lcd, potentiometer)
Web integration	Limited / indirect	SSE server streams live notes to Angular app

Technology stack (arduino vs ESP32)

3. AR integration (Unity + Vuforia) — Piano tutor mode

This chapter presents the augmented reality (AR) module of Piano, designed as a tutor to help users learn melodies with step-by-step guidance. The system combines real-time ESP32 note detection with visual overlays in Unity, providing multimodal feedback (visual, text, and tactile) to make learning more intuitive and engaging.

3.1 Purpose and learning goals

The AR tutor helps users recognize and play notes on the physical piano by overlaying visual hints directly onto the keys. The camera tracks the instrument as a target, and Unity displays guidance in real time. Feedback rules are straightforward: the next correct key is highlighted green, while any incorrect key pressed turns red temporarily. A sheet music sequence is displayed in the UI, allowing users to follow melodies and monitor their progress.

This approach is particularly beneficial for beginners, older adults, and users with disabilities. By providing visual guidance in combination with auditory and tactile feedback, the system reduces cognitive load and makes learning accessible to a wide range of users.

3.2 Technologies and architecture

3.2.1 AR tracking

Vuforia Engine is used to recognize the piano as a Model or Image Target. Targets are created with the Model Target Generator and imported into Unity via Vuforia databases. Runtime tracking relies on event callbacks, such as OnTargetFound and OnTargetLost, which trigger the AR tutor’s behavior.

3.2.2 Hardware-to-Unity communication

The ESP32 streams note data to Unity via a serial connection (COM5/COM6) at 115200 baud. Each note is sent as a text line (for example, C4, C#4, D4) and parsed by Unity to trigger visual actions on the virtual piano overlay.

3.2.3 Unity UI

The user interface is built with TextMeshPro on a Canvas. It displays the current partitura, object information, and buttons for toggling between general descriptions and usage instructions for recognized 3D objects.

3.3 Main Unity components (scripts)

EspSerialReader acts as a bridge between ESP32 and Unity. It reads serial input line by line, normalizes note messages, and calls piano.SimulatePressByName(note) for each detected key. This module can start or stop based on target visibility events.

ButtonsColoring implements the tutor logic. Each of the twelve virtual key spheres is mapped to its corresponding note in octave 4. The component highlights the next correct key in green, advances the sequence when the right note is played, and shows a temporary red highlight if a wrong key is pressed. The “partitura” string is exposed to the UI through events to update text instantly.

PartituraLabel listens for updates from ButtonsColoring and displays the current melody string, such as Partitura: C4 C#4 B4 ....

Objects3DChanger and TextChanger enhance the AR experience by providing information and instructions for each recognized object. Users can toggle between general descriptions and usage guidance for the active object.

3.4 Runtime Flow

When the AR scene starts, Vuforia attempts to detect the piano target. Once found, OnTargetFound triggers the ButtonsColoring.RestartSong() method, which highlights the first note and updates the partitura in the UI. As the ESP32 streams note events over serial, Unity validates each input:

- ➢ Correct note: green highlight advances to the next key.
- ➢ Incorrect note: pressed key turns red for approximately two seconds, then resets; the correct target remains green.

If tracking is lost, OnTargetLost clears the UI and state, and user input is ignored until the target is detected again.

3.5 Common Errors and Troubleshooting

Some known issues and their fixes include:

i)Target database not detected (after reopening the project)

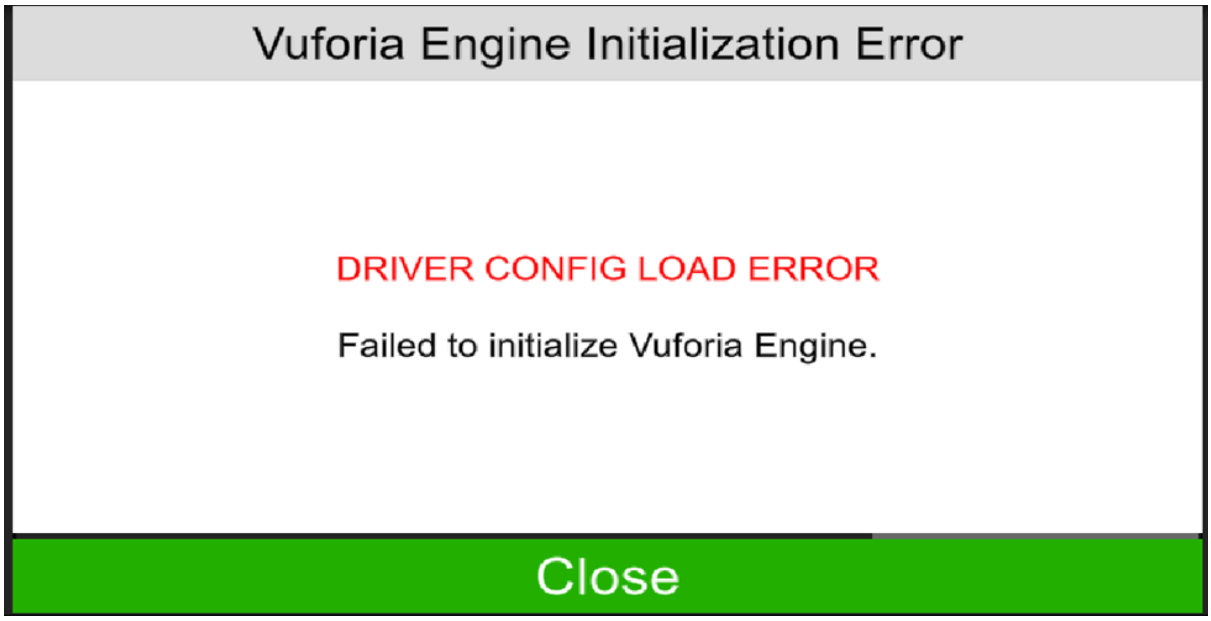


Fig 13

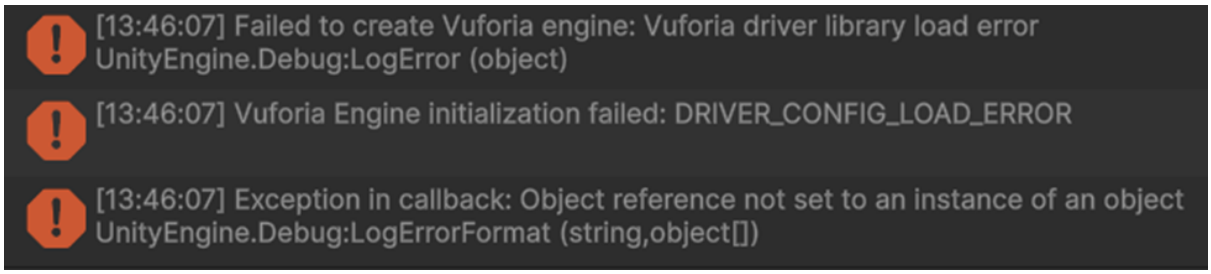


Fig 14

Sometimes Vuforia fails to detect targets after closing and reopening the project. What you can do in order to fix it:

- ➔ remove the image/model from the target object and re-add it,
- ➔ save the scene afterward.

➔ if the issue persists, re-entering the license key can resolve it.

ii)Multiple model targets active

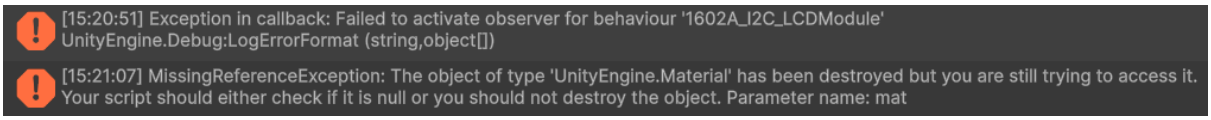


Fig 15

When two or more Model Targets are active simultaneously, an observer-related error can occur. The project should ensure only one Model Target is active at runtime.

iii)Unity instability

Under high load or when triggering unimplemented actions, Unity may freeze/crash. The practical mitigation is frequent saving and incremental testing.

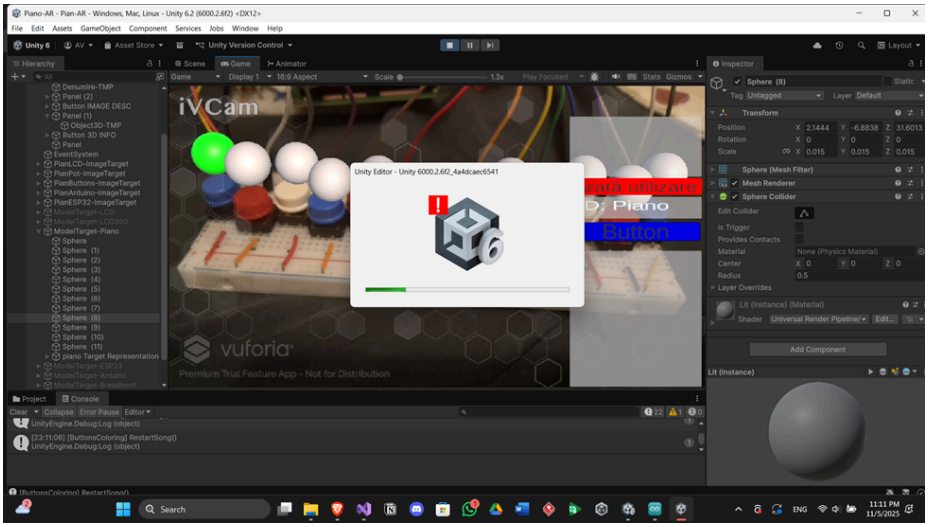


Fig 16

3.6 Limitations (Current Constraints)

- *Model targets and area targets require a premium license* - without it, Vuforia displays a watermark. Hiding Model Targets can remove the watermark, but it affects AR functionality

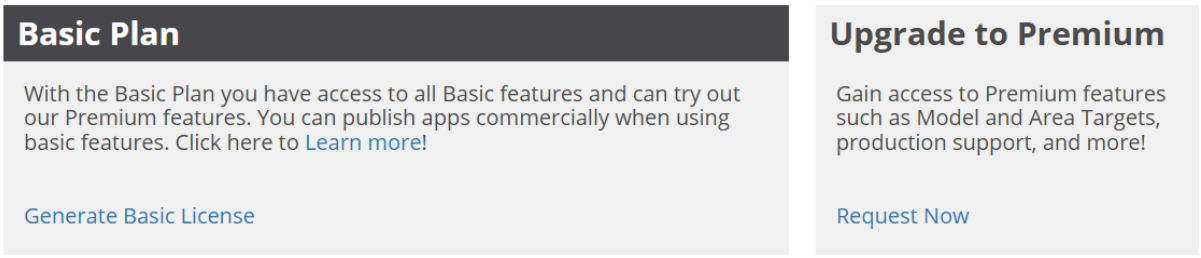


Fig 17

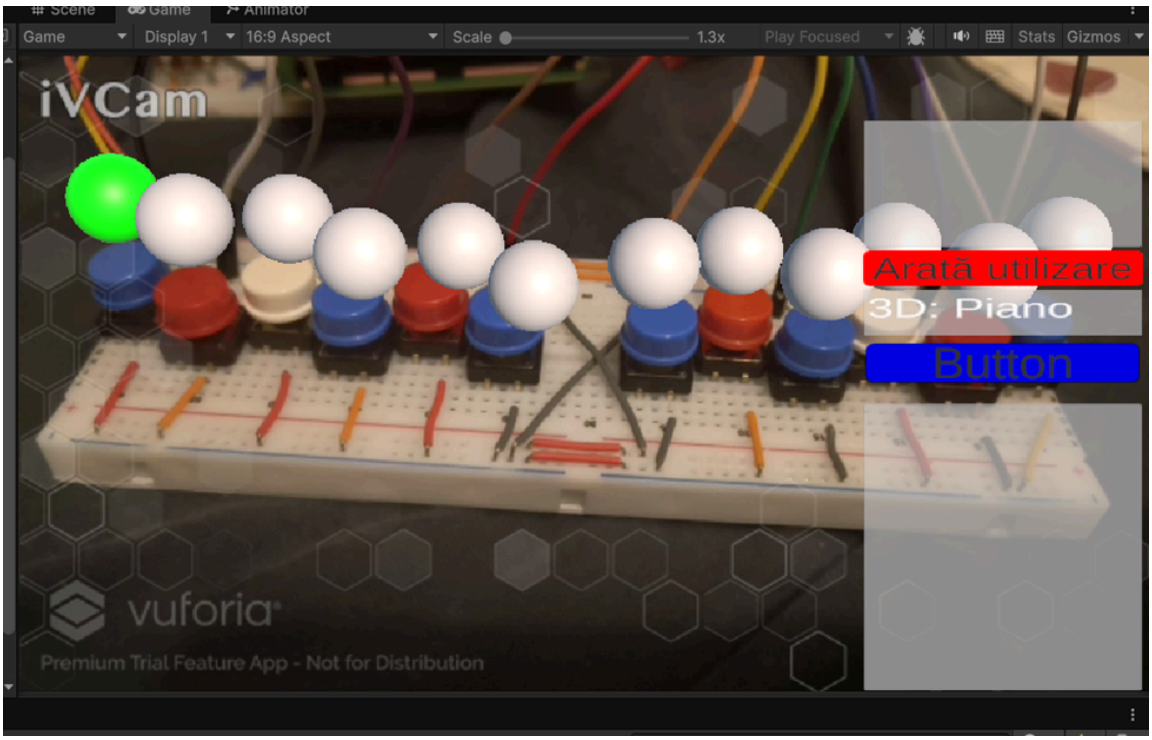


Fig 18

- Some *targets are more complex and are recognized less reliably*, which can prevent overlays from appearing consistently.

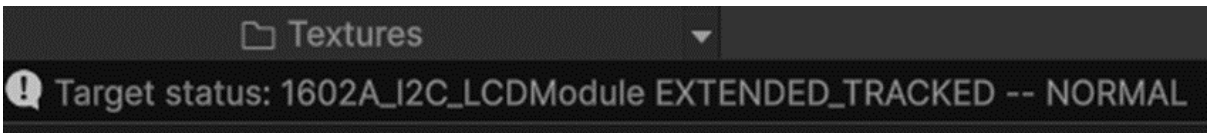


Fig 19

- *Training model targets* can take 2–4 hours, depending on whether the model is stationary or 360° vision.

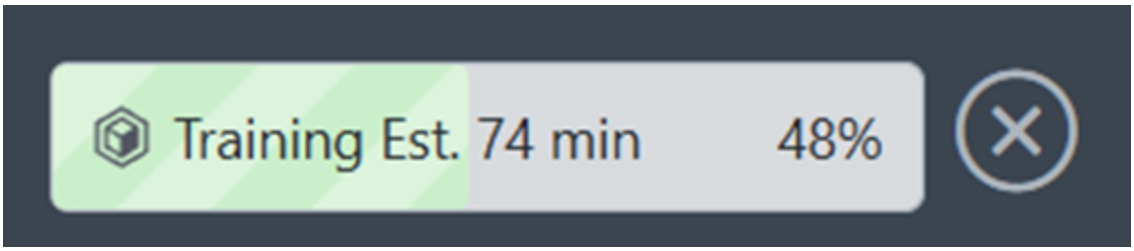


Fig 20

3.7 Future improvements (next iterations)

Several enhancements are planned for future iterations of the AR module to improve usability, reliability, and the learning experience. These improvements include:

➢ Add score, accuracy, and time-per-note metrics. This will allow users to receive quantitative feedback on their performance, helping them track progress and focus on areas that need improvement.

➢ Implement automatic COM reconnection and a "READY" handshake message from the ESP32. These features will ensure more reliable communication between the AR system and the piano, reducing interruptions during practice.

➢ Introduce bidirectional communication from Unity to the ESP32. This will allow the AR system to send real-time status updates or guidance directly to the physical LCD, further integrating the virtual and physical learning experience.

3.8 Demo

The AR-Piano demo showcases real-time visual guidance without audio. Users can follow melodies visually while the system validates key presses in real time.

AR-Piano Demo

4. Azure AI melody recognition (ESP32 → HTTPS → API → AI → LCD)

The Azure module transforms the ESP32 piano from a simple note generator into a connected IoT assistant. Instead of only playing tones, the system can record a short melody, send it to a cloud API hosted on Azure, and receive back the closest matching song name. This result is then displayed to the user on the 16×2 LCD, keeping the interaction intuitive and accessible.

4.1 Hardware modification for Azure integration

The Azure feature requires only a minimal hardware change: an additional Record/Play button connected to GPIO26. All other components remain unchanged, including:

- The 12 piano keys (some connected directly to GPIO pins, others via PCF8574 I²C expander)
- Piezo buzzer for audio feedback
- LCD 16×2 for textual feedback
- Potentiometer for user-controlled adjustments
- ESP32 Wi-Fi for cloud connectivity

This ensures that the physical piano setup is mostly unchanged, while enabling cloud-based functionality.

.

4.2 User interaction flow

Recording and sending a melody is handled through a two-press logic using the single Record/Play button:

- Press GPIO26 once → Start recording
    - is_recording is set to true

- The note buffer is cleared (recorded_count = 0)
- Recording timestamp begins (record_start_time)
- Key presses create entries containing note index (0–11), computed frequency, relative start time, and duration

➢ Press GPIO26 again → Stop recording and send to Azure

- is_recording is set to false
- is_playing_back is set to true (local playback mode)
- send_to_azure is set to true, signaling the dedicated Azure task to send the recorded melody
- The API returns a JSON response containing the closest song name, which is parsed and displayed on the LCD

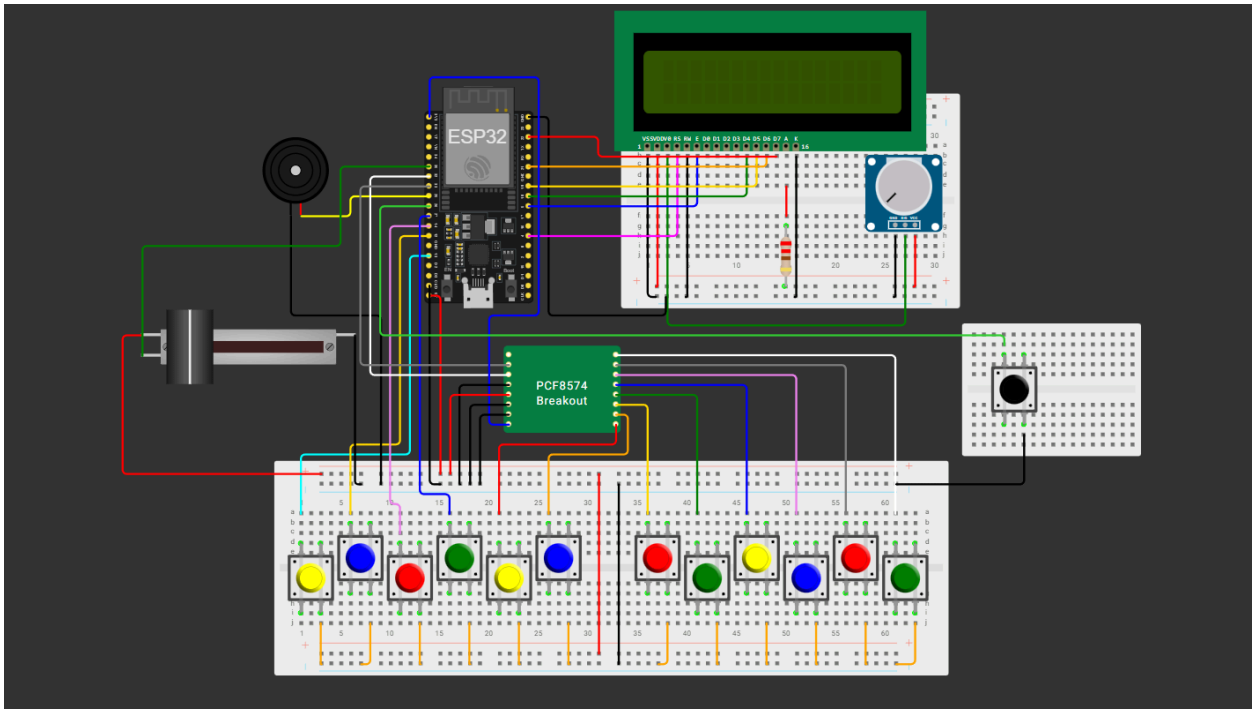


Fig 21 : Hardware scheme with ESP32 for Azure

4.3. User interaction flow (two-press logic)

The Azure feature is designed to be controlled using a single physical button, making it simple for users to record and send melodies. The workflow is based on a two-press logic.

When the user presses the Record/Play button (GPIO26) for the first time, the ESP32 enters recording mode. At this point, the firmware sets the is_recording flag to true, clears the note buffer (recorded_count = 0), and starts the recording timestamp (record_start_time). As the user plays notes on the piano, each key press is stored in the buffer as a separate entry, which includes:

- ➢ The index of the pressed note (0–11)
- ➢ The computed frequency of the note
- ➢ The start time relative to the beginning of the recording
- ➢ The duration of the note, calculated when the key is released

Once the user presses the button a second time, recording stops. The firmware sets is_recording to false and enables local playback mode (is_playing_back = true). Most importantly, the send_to_azure flag is set to true, signaling a dedicated task to transmit the melody to the Azure API. After this process, the firmware receives a JSON response from the cloud, parses it, and extracts the detected song name, which can then be displayed on the LCD.

4.4. Why a dedicated azure task exists

Network communication, especially over HTTPS with SSL, requires additional memory and can block the main execution of the firmware. To avoid interrupting real-time piano interactions, a dedicated Azure task handles cloud requests independently.

- ➢ The Azure_task() is allocated a large stack (AZURE_TASK_STACK_SIZE = 8192) to accommodate network operations.
- ➢ It continuously waits for the send_to_azure flag to become true.
- ➢ Once triggered, it calls send_melody_to_ai() to send the recorded melody and retrieve the AI response.

This architecture ensures that key scanning, buzzer output, and LCD updates continue to run smoothly, even during network operations.

4.5. Wi-Fi and HTTPS connection

The ESP32 connects to the network in Station mode using the wifi_init() function. This process:

- ➢ Initializes NVS, network interfaces, and the event loop
- ➢ Starts the Wi-Fi driver and automatically reconnects if disconnected
- ➢ Prints a confirmation message when the IP address is obtained

For secure cloud communication, HTTPS is configured using:

- ➢ esp_tls_init_global_ca_store() to set up the certificate store
- ➢ An HTTP client with SSL transport (HTTP_TRANSPORT_OVER_SSL)
- ➢ A 15-second timeout (timeout_ms = 15000)
- ➢ skip_cert_common_name_check = true to simplify development while testing

4.6. JSON payload sent to the API

When sending a melody to Azure, the firmware constructs a JSON payload using cJSON. The payload contains the following fields:

- ➢ deviceId: "piano_esp32"
- ➢ message: a human-readable string showing the recorded melody and total number of notes
- ➢ sensorType: "melody"
- ➢ value: recorded_count (total notes recorded)
- ➢ unit: "notes"

The melody sequence is embedded in the message field, with note names joined by hyphens and limited to 20 notes to keep the payload compact. For example:


```
{
  "deviceId": "piano_esp32",
  "message": "Melody recorded: C4-D4-E4-G4 (Total 4 notes)",
  "sensorType": "melody",
  "value": 4,
  "unit": "notes"
}
```


This ensures that the payload is concise and interpretable by the cloud API.

4.7. API call to Azure endpoint

The firmware sends the JSON payload to the Azure endpoint via an HTTPS POST request. During this process:

- ➢ The URL is set to the API endpoint (https://<azurewebsites>.net/api)
- ➢ Headers include Content-Type: application/json
- ➢ The JSON body is generated using cJSON_Print(json)

After sending the request, the firmware prints both the HTTP status code and the raw response, which is useful for debugging.

4.8. Parsing the AI response

Once the response is received, the firmware parses the JSON buffer using cJSON_Parse(). The "response" field contains the closest song name, which the ESP32 extracts with cJSON_GetObjectItem(). This song name is intended for display on the LCD.

In the current implementation, the firmware prints:

Detected song: <song name>

In the fully integrated system, this string can be displayed on the 16×2 LCD for the user.

4.9. AI instructions and behavior

On the Azure side, the AI model is guided by a fixed instruction that defines both its role and the expected output format. The instruction enforces a constrained and deterministic response, suitable for embedded systems with limited display capabilities.

The AI is instructed to behave as a music expert and to analyze the received melody sequence. Based on the provided notes, it must identify the closest matching known song. The response must contain only the song name, without explanations or additional text.

The instruction used for the AI model is the following:

"You are an expert in music. You respond with the sequence of notes that was sent and the closest song based on the sequence of notes given by the user. You only respond with the song name and nothing else."

This strict output constraint is essential because the ESP32 firmware expects a short string that can be displayed directly on the 16×2 LCD without further processing or filtering.

4.10 Demo

For direct acces to live demo, please refer to the following link: Azure Demo

5. Web Module: Piano-Web (Angular) — Live Companion for the Physical Piano

5.1 Overview

Piano-Web is a web-based companion application developed in Angular, designed to complement the physical piano. Unlike a virtual piano that simulates note generation solely in the browser, Piano-Web is driven by real hardware. Each physical key press on the piano is transmitted in real time by the ESP32 to the web interface. This approach allows the browser to reflect the actual state of the instrument, providing a live, interactive learning environment.

The module enhances the user experience by offering both immediate visual feedback and guided tutoring for selected melodies, transforming the system into an accessible and educational platform.

5.2 Communication Architecture

Piano-Web receives input from the physical piano using **Server-Sent Events (SSE)**. The ESP32 exposes an HTTP endpoint that maintains a persistent connection to connected browsers. Each key press generates an SSE event that is broadcast to all clients, enabling the browser to update the user interface without polling.

SSE is particularly suitable for this system because it supports one-way streaming, uses standard HTTP protocols, and allows browsers to maintain persistent connections through the built-in EventSource API.

The live data flow proceeds as follows:

1. The user presses a physical piano key.
2. The ESP32 detects the input via GPIO or the PCF8574 I²C expander.
3. An SSE event is sent to all connected browsers.
4. Angular processes the event, updating the UI by highlighting the pressed key, displaying the last note, and checking progress if Melody Mode is active.

This architecture ensures that the web interface mirrors the physical instrument accurately and provides immediate feedback for each action.

5.3 Web Modes

Piano-Web supports two distinct modes to accommodate both exploration and structured learning:

5.3.1 Free Play Mode

In Free Play Mode, the web application functions as a live visual mirror of the physical piano. The UI highlights the currently pressed key, displays the most recent note, and may optionally show a history of previous presses. No correctness checks are performed, allowing users to freely explore the instrument.

This mode is particularly useful for testing the system's connectivity and for allowing users to experiment with the piano without constraints.

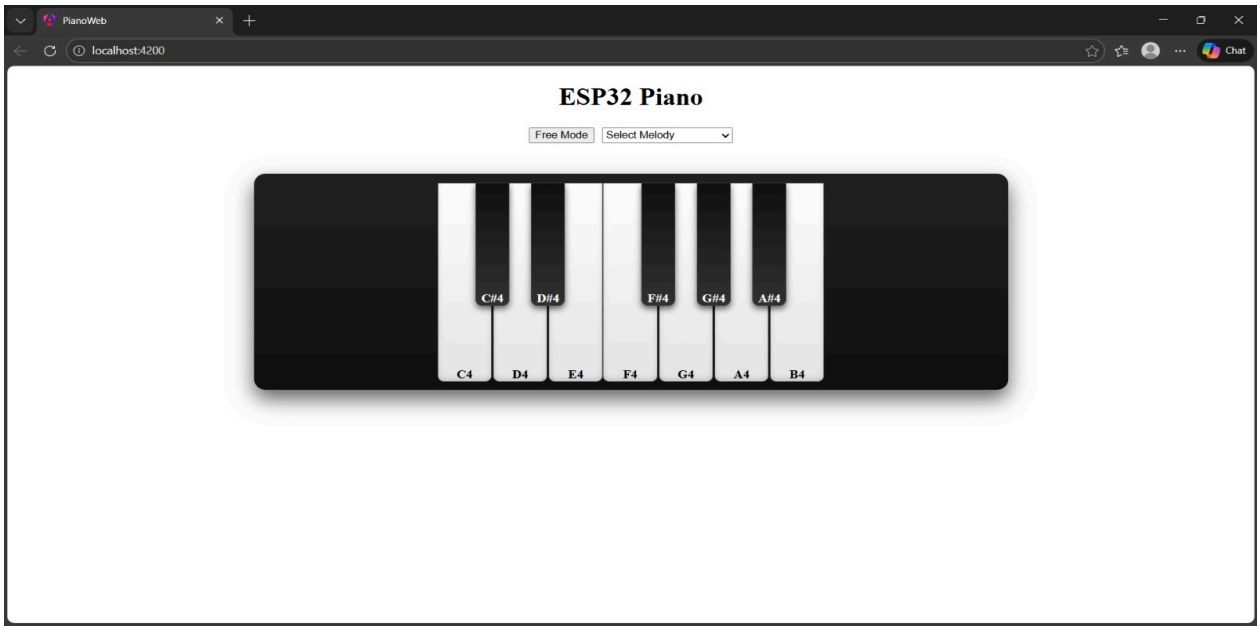


Fig 22: Web interface

5.3.2 Melody Mode

Melody Mode provides guided instruction through predefined sequences of notes. Users select a melody, and the application sets an internal step index to track progress. The interface highlights the next expected key, and as notes are played on the physical piano, the system evaluates correctness. Correct presses advance the melody, while incorrect presses trigger temporary visual feedback without penalizing the user.

This tolerant design supports learning and accessibility by:

- ➢ Allowing beginners to correct mistakes without restarting the sequence.
- ➢ Reducing cognitive load for older adults.
- ➢ Enabling users with motor impairments to participate effectively.

Fig 23: Web interface

5.4 Angular Responsibilities

The web application implements the following core functions:

- ➢ handleNote(): Processes incoming note events from the ESP32.
- ➢ startMelody(notes): Loads a melody and initiates the guided session.
- ➢ resetMelody(): Resets progress in the current melody.
- ➢ switchToFreeMode(): Returns the UI to Free Play Mode.
- ➢ isActive() / isHighlighted(): Manages key highlighting for correct and incorrect presses.

5.5 Accessibility Considerations

Piano-Web is designed to improve accessibility and support human-computer interaction principles. Key features include:

- ➢ Clear visual cues and large interactive elements to reduce cognitive load.
- ➢ Tolerant error handling to encourage experimentation and learning.
- ➢ Immediate feedback linking physical action to visual and musical response.

The combination of a physical instrument with an adaptive web interface provides an effective educational platform for a broad audience, including beginners, older adults, and users with disabilities.

5.6. Demo

For direct access to the live demo, please refer to the following link: WebDemo

6. Final demonstration: all features

The final demonstration highlights the complete Piano system, integrating all implemented functionalities. This includes the ESP32-based IoT piano, the recording feature with Azure AI melody recognition, the AR tutor overlay for guided learning, and the web-based melody mode for step-by-step practice.

The demonstration allows viewers to observe how the hardware, cloud services, AR interface, and web companion interact seamlessly, providing a comprehensive overview of the system's capabilities.

For direct access to the full demonstration, please follow this link: FinalDemo

7. Project source code (GitHub)

The complete source code for the Piano project is available on GitHub. It includes all modules of the system (ESP32 firmware, Azure API integration, AR tutor, and the web companion application), along with the required configuration files and documentation : GitHub Link