Technical Report for Python Final Project

Speech-to-Text Notes App

Team Members: Cassidy Sakamoto, Justin Glabicki, Nathan Tan
Course: EECE 2140 – Computing Fundamentals for Engineers
Instructor: Dr. Fatema Nafa

December 5, 2025

Contents

Abstract

Students, professionals, and people with disabilities often need to capture dense technical information during lectures, labs, meetings, and more. Traditional handwritten notes can be incomplete, difficult to organize, and hard to search later. Our project address this problem by implementing a desktop Speech-to-Text Notes App that records audio from a microphone, transcribes spoken language into text, and organizes the resulting notes with titles, tags, timestamps, and export options. The system is implemented entirely in Python. A microphone backend based on the sounddevice library captures mono audio at a fixed sampling rate in small frames. These frames are aggregated into multi-second chunks and sent to a transcription pipeline implemented with the speech_recognition library and the Google Web Speech API. We apply light text normalization, including domain-specific replacements for technical terms (e.g. PWM, Hz) and simple punctuation handling. A Tkinter graphical user interface provides start/stop recording controls, a live mic-level meter, timestamp markers a searchable notes list, and export to .txt and .md files backed by JSON storage. Our results show that the app can record and transcribe continuous speech in real time while allowing users to edit, tag, and search their notes without losing data due to autosave and persistent storage. The project demonstrates modular program design, real-time audio handling, and GUI programming in Python. During the creation of this app, there were some limitations. Two of these limitations included network dependence and basic error handling. Future work includes offline transcription models, richer formatting, and more advanced search features.

1 Introduction

Taking high-quality notes is critical in a student and professional context, especially including and engineering context. Lectures, labs, and project meetings often introduce many equations, acronyms, and technical details in a short time. Manually writing everything down can cause students to miss context, especially when diagrams or live demonstrations are involved. A speech-to-text notes application can reduce the cognitive load of note-taking by automatically transcribing spoken content into searchable text that can be edited, tagged, and organized afterwards. Our project implements a Python-based Speech-to-Text (STT) Notes App that focuses on three main goals: reliable real-time capture and chunking of audio, robust transcription with basic error handling and domain-specific normalization, and a user interface that feels like a practical note-taking tool instead of a demo script. The app is aimed to use cases such as recording lectures, review sessions, or personal study sessions where the user wants a running transcript plus the ability to annotate, insert time markers, and export final notes. From a computing perspective, this project combines several core topics: modular design (separating backend audio/transcription from frontend UI), data structures and JSON persistence, event-driven programming with callbacks and timers, and basic multithreading to keep real-time audio capture responsive while long-running tasks like network transcription run in the background.

1.1 Student Learning Objectives

After completing this project, students should be able to:

- Design and implement modular Python programs that separate concerns between back-end logic and a graphical user interface.
- Translate a high-level specification (”record audio, transcribe, manage notes”) into concrete data structures, classes, and functions.
- Apply appropriate data structures (lists, dictionaries, dataclasses) to manage in-memory state and persistent storage.
- Write pseudocode and technical documentation that explain the flow of control, including event callbacks and autosave logic.
- Evaluate behavior of a real-time system and interpret the output under different conditions (quiet vs. noisy environment, different speaking styles).

1.2 Team Objectives

Our team’s specific objectives for this project were:

- Build a complete python application that can record microphone audio, stream it to a transcription backend, and display recognized text to the user.
- Use clean, modular code: separate the microphone input class, transcription logic, data model, and UI unto distinct components with clear interfaces.
- Implement note management features including titles, tags, search/filtering, autosave, and export to plain text and markdown files.
- Provide visual feedback for microphone level and recording duration, making it clear when the system is listening and how loud the input is.
- Validate correctness and robustness using simple test cases (e.g. saving/loading, handling empty transcripts, and long recording sessions).

1.3 Report Structure

The remainder of this report is organized as follows. The System Diagram section presents a high-level system diagram of the STT Notes App architecture. The Methodology section describes the methodology and program architecture, including pseudocode for the main components. The Data Structures Used section discusses the data structures used and why they were chosen. The Experimental Setup section explains the experimental setup and test cases. The Results and Analysis section presents the results and analysis of the app’s behavior. The What We Learned section summarizes what we learned, and the Conclusion and Future Work section concludes with possible future work. The appendices contain the full source code and any additional figures or tables.

1.4 System Diagram

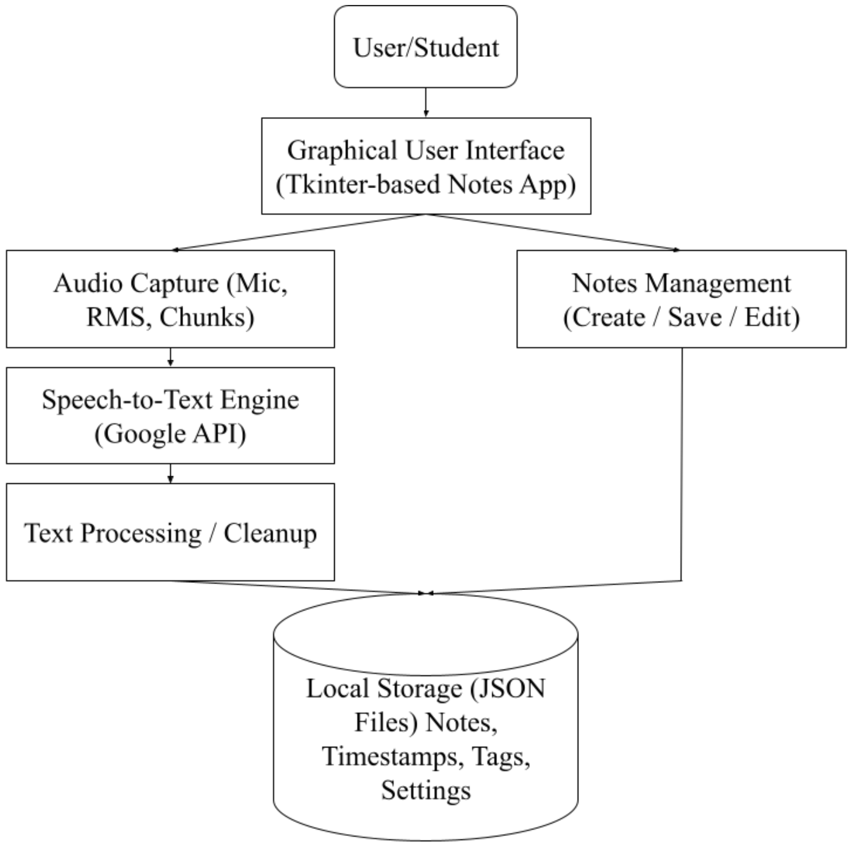

User/Student
Graphical User Interface (Tkinter-based Notes App)
Audio Capture (Mic, RMS, Chunks)
Notes Management (Create / Save / Edit)
Speech-to-Text Engine (Google API)
Text Processing / Cleanup
Local Storage (JSON Files) Notes, Timestamps, Tags, Settings

Figure 1: High-level system architecture of the system architecture of the Speech-to-Text Notes App.

2 Methodology

Our solution is divided into four main components:

1. A **microphone input** class that uses sounddevice to stream audio from the default input device.
2. A **transcription pipeline** that aggregates short frames into chunks, calls the Google Web Speech API, and normalizes the recognized text.
3. A **data model and persistence layer** using a Note dataclass and JSON files for saving and loading notes.
4. A **Tkinter GUI** that manages recording controls, note editing, search, tags, autosave, and export, while also visualizing microphone level.

The system is event-driven: audio callbacks fire when new samples arrive, background threads perform transcription to avoid blocking real-time capture, and the UI updates via periodic timers and user actions (button clicks, keyboard shortcuts).

2.1 Program Architecture

At a high level, the program is organized into the following modules and classes:

- **MicrophoneInput**: encapsulates the sounddevice.InputStream. It computes a root-mean-square (RMS) level for each incoming chunk and forwards the audio to a callback function for further processing.
- **Transcription Functions**: top-level functions such as process_audio, transcribe_chunk, recognize_google_raw, and normalize_text implement chunk aggregation, retry logic, calls to the recognizer, and domain-specific text cleanup.
- **Data Model**: a @dataclass called Note stores each note's ID, title, content, tags, creation time, last update time, and optional recording duration. Helper function load_notes and save_notes handle JSON persistence.
- **Config and Dictionary**: additional JSON files store configuration values (e.g. autosave delay, noise gate threshold) and an optional list of domain-specific words edited in the UI.
- **App (Tkinter)**: the main App class derives from tk.Tk. It builds the layout (sidebar listbox, search bar, buttons, text editor, status bar), wires UI events to backend actions, and implements autosave and export. It also registers a global transcript_callback so transcription results can be appended to the current note.

The flow of execution is:

1. The user launches the app, which loads any existing notes and configuration from JSON.
2. When the user presses "Start Recording," the MicrophoneInput stream is started and begins delivering audio chunks.
3. Each chunk is converted to 16-bit PCM bytes and buffered; once enough frames have been collected for a chunk, a background thread calls the transcription API.
4. The normalized transcript text is sent back to the UI via transcript_callback, which inserts it into the editor.
5. Edits to the note (content, title, tags) are autosaved for a short delay; the user can also manually save or export at any time.

2.2 Pseudocode for Main Components

Provide pseudocode for each important part of your project.

Algorithm 1 STT Notes App - High-Level Flow

```
    Part 1: Microphone Input
 1: function StartRecording
 2:     recStartMs ← currentTimeMs()
 3:     elapsedMs ← 0
 4:     configure MicrophoneInput with RMS threshold and sample rate
 5:     start sounddevice input stream with callback AudioCallback
 6: end function
 7: function AudioCallback(audioChunk)
 8:     rms ← computeRMS(audioChunk)
 9:     updateMicLevel(rms)                              ▷ for level meter in UI
10:     pcmBytes ← convertFloatToInt16(audioChunk)
11:     append pcmBytes to global frameBuffer
12:     if frameBuffer has enough frames for one chunk then
13:         rawChunk ← concatenate frames in frameBuffer
14:         clear frameBuffer
15:         launch background thread TranscribeChunk(rawChunk)
16:     end if
17: end function

    Part 2: Transcription and Normalization
18: function TranscribeChunk(rawChunk)
19:     for attempt from 1 to maxRetries do
20:         text ← callSpeechAPI(rawChunk)
21:         if text recognized then
22:             break
23:         else if network error and attempt < maxRetries then
24:             sleep briefly and retry
25:         else
26:             log warning and return
27:         end if
28:     end for
29:     cleanText ← NormalizeText(text)
30:     send cleanText to UI via transcriptCallback
31: end function
32: function NormalizeText(text)
33:     t ← lowercase(text)
34:     apply domain replacements (“p w m” → “PWM”, etc.)
35:     collapse extra whitespace
36:     if t does not end with punctuation then
37:         append “.” to t
38:     end if
39:     capitalize first character
40:     return t
41: end function
```

Algorithm 2 STT Notes App - High-Level Flow

```
Part 3: Notes Management and UI
function AppendTranscriptToNote(cleanText)
    if editor is not empty then
        insert space in text widget
    end if
    insert cleanText at end of text widget
    record lastEditTime $\leftarrow$ currentTimeMs()
end function
function AutosaveTick
    if autosaveEnabled and noteChangedRecently then
        SaveNote(viaAutosave = true)
    end if
    schedule next AutosaveTick in 100 ms
end function
function SaveNote(viaAutosave)
    title $\leftarrow$ titleEntry or default timestamp
    tags $\leftarrow$ split tagsEntry on commas
    content $\leftarrow$ full text from editor
    if note has existing id then
        update corresponding Note in notes list
    else
        create new Note with new UUID and timestamps
    end if
    write all Notes to JSON file
    update status bar with time of save
end function
```

3 Data Structures Used

We used several standard Python data structures and the dataclasses module to keep the code simple and modular:

- **Lists**:
  - A list of byte frames (frames) buffers short audio frames until a full chunk is ready.
  - A list of Note objects (self.notes) stores all notes in memory for filtering and display.
  - Lists are also used to store domain words and tags.
- **Dataclasses**:
  - The Note dataclass groups fields such as id, title, content, tags, createdAt, updatedAt, and durationMs into a single type that can be easily serialized to and from JSON.

- **Dictionaries**:
  - A configuration dictionary stores settings like autosave delay, noise gate threshold, chunk length, and retry count.
  - Another dictionary (DOMAIN_REPLACE) maps domain phrase patterns (e.g."p w m") to normalized technical tokens (e.g. "PWM").
- **Strings and Lists of Strings**:
  - Tags are stored as lists of strings so that we can join/split them for display and filtering.
  - Search queries and note contents are treated as strings and scanned linearly for filtering.

These choices are appropriate for several reasons:

- **Time complexity**: Appending audio frames and notes to lists is amortized $O(1)$, and iterating through the list of notes for filtering is $O(n)$, which is acceptable given the expected note counts in a personal notes app.
- **Space efficiency**: Lists and dictionaries are memory-efficient and built into Python. JSON serialization of lists of dictionaries (derived from dataclasses) is straightforward.
- **Readability and modularity**: Using a dataclass for Note makes it clear what fields belong to a note object, improving readability and making it easy to extend the model later (e.g. adding a "favorite" flag).

4 Experimental Setup

To evaluate the performance and behavior of the Speech-to-Text Notes App, we conducted a series of tests using different audio input conditions, chunk lengths, noise gate thresholds, and note lengths. The system was executed both in a standard Python environment and inside a Jupyter Notebook, using the Tkinter GUI for all user interactions.

Hardware and Environment

- **Operating System:** Windows 11
- **Microphone:** Internal laptop microphone
  (Microphone Array – Intel® Smart Sound Technology)
- **Execution environment:** Tkinter GUI launched from:
  - Jupyter Notebook (for early prototyping)
  - Standard Python interpreter (for full application testing)
- **Python Version:** 3.x

Software Dependencies

The following libraries were used in the backend and GUI:

- `sounddevice` – real-time microphone capture
- `numpy` – numerical operations and PCM conversion
- `speech_recognition` – Google Web Speech API interface
- `tkinter` – graphical user interface
- `json`, `time`, `threading`, `struct`, `uuid` – application utilities

Parameters Tested

We evaluated system performance across three main categories:

1. **Noise Gate Thresholds (Silence Filtering)**
2. **Chunk Lengths (Transcription Latency vs. Stability)**
3. **Input Note Length (Short vs. Long Speech Samples)**

Noise Gate Threshold Tests

To determine how sensitive the transcription pipeline is to quiet speech and background noise, we tested the following thresholds:

- **0.03** – high threshold (aggressive filtering)
- **0.002** – medium threshold
- **0.0** – noise gate disabled

Each threshold was evaluated using a fixed chunk length of **200 ms**. The following script was used for the tests: "An oscilloscope is an electronic instrument that visually represents an electrical signal as a graph, with voltage on the vertical axis and time on the horizontal axis."

Chunk Length Tests

To understand how chunk duration affects latency, responsiveness, and transcription accuracy, we tested:

- **200 ms** – fastest updates, highest API call frequency
- **1000 ms** – moderate delay
- **2000 ms** – slower updates, fewer API requests

These chunk lengths were tested using a fixed noise gate threshold of **0.002**. The following script was used for the tests: "A wind tunnel is a powerful device that creates a controlled stream of air to test how objects like aircraft, cars, or buildings react to wind."

Input Scripts (Short vs. Long Notes)

The following spoken scripts were used to test transcription behavior under short and long note-taking conditions:

- **Short script:**
  "Hello. This is test one."
- **Long script:**
  "B92 was initially referred to as 'the black hole,' given its appearance after it was first catalogued in 1913. It was later discovered to be a dark nebula, and the title is now misleading, as the name black hole is used in modern astrophysics to describe a region of spacetime in which gravity is too strong for light to escape."

Unless otherwise specified, these scripts were tested with a noise gate threshold of **0.002** and a chunk length of **2000 ms**.

Test Procedure

For each test configuration, we evaluated:

- Transcription accuracy for normal speech
- Response delay between speaking and text appearing in the UI
- Behavior during quiet periods or background noise conditions
- Performance differences between short and long speech samples
- Stability of the application over extended recording sessions

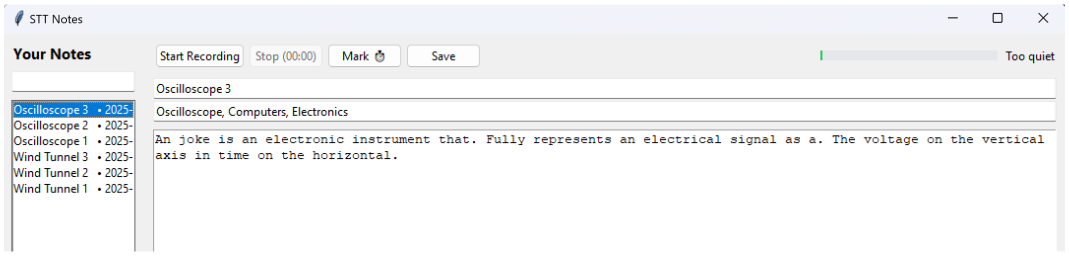
5 Results and Analysis

This section presents the outcomes of our speech-to-text system tests across three main categories: noise gate thresholds, chunk lengths, and note length (short vs. long input). Each subsection includes screenshots of representative outputs along with a detailed interpretation of the system's performance.

5.1 Noise Gate Threshold Results

To understand how noise suppression impacts transcription, we tested three thresholds: 0.03, 0.002, and 0.0. The following figures show representative outputs from each test.

Threshold = 0.03 (High Filtering)



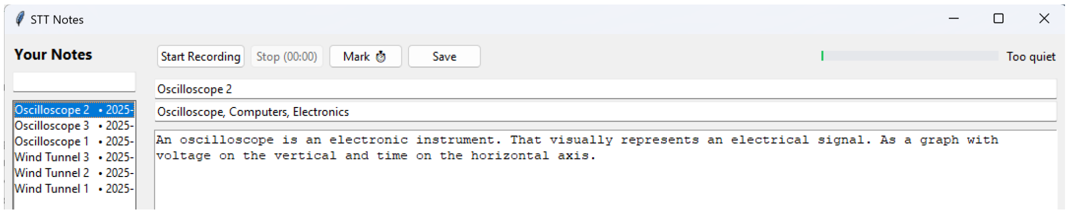
Oscilloscope 3 • 2025-
Oscilloscope 2 • 2025-
Oscilloscope 1 • 2025-
Wind Tunnel 3 • 2025-
Wind Tunnel 2 • 2025-
Wind Tunnel 1 • 2025-

Oscilloscope 3
Oscilloscope, Computers, Electronics

```
An joke is an electronic instrument that. Fully represents an electrical signal as a. The voltage on the vertical
axis in time on the horizontal.
```

Analysis: At this threshold, the system treated many speech frames as silence, resulting in missing or inaccurate words . This matched expectations because most normal speech RMS values fall below 0.03 on laptop microphones.

Threshold = 0.002 (Medium Filtering)



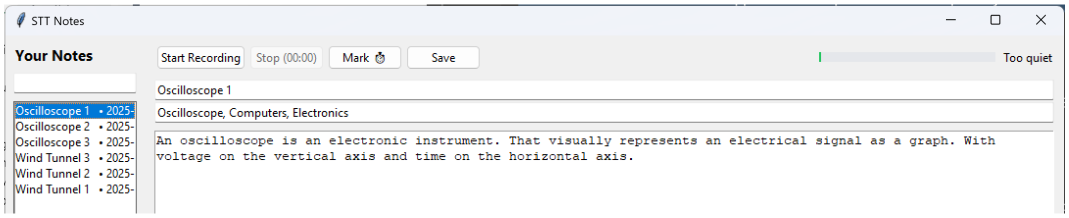
Oscilloscope 2 • 2025-
Oscilloscope 3 • 2025-
Oscilloscope 1 • 2025-
Wind Tunnel 3 • 2025-
Wind Tunnel 2 • 2025-
Wind Tunnel 1 • 2025-

Oscilloscope 2
Oscilloscope, Computers, Electronics

```
An oscilloscope is an electronic instrument. That visually represents an electrical signal. As a graph with
voltage on the vertical and time on the horizontal axis.
```

Analysis: This threshold provided a balance between noise suppression and speech detection. Clear speech mostly passed through while background noise was reduced. This setting produced the most consistent transcription performance. One example of a word being cut out is the first time axis is mentioned in the script.

Threshold = 0.0 (Noise Gate Disabled)



Oscilloscope 1 • 2025-
Oscilloscope 2 • 2025-
Oscilloscope 3 • 2025-
Wind Tunnel 3 • 2025-
Wind Tunnel 2 • 2025-
Wind Tunnel 1 • 2025-

Oscilloscope 1
Oscilloscope, Computers, Electronics

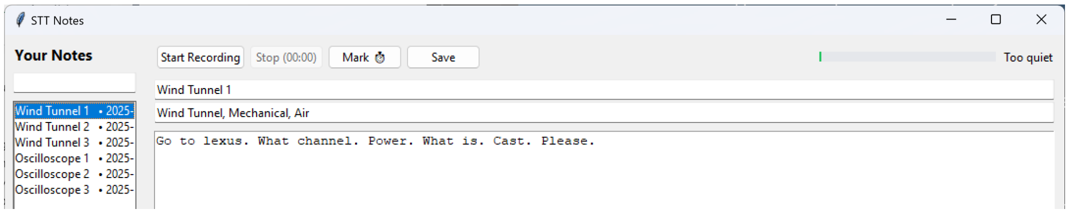
```
An oscilloscope is an electronic instrument. That visually represents an electrical signal as a graph. With
voltage on the vertical axis and time on the horizontal axis.
```

Analysis: With the noise gate disabled, all audio frames were forwarded, resulting in complete transcripts but occasionally including background noise or inaccuracies were transcribed. This served as a useful baseline for comparison.

5.2 Chunk Length Results

We next evaluated how varying chunk lengths affected transcription latency and accuracy. Three chunk durations were tested: 200 ms, 1000 ms, and 1000 ms.

Chunk Length = 200 ms



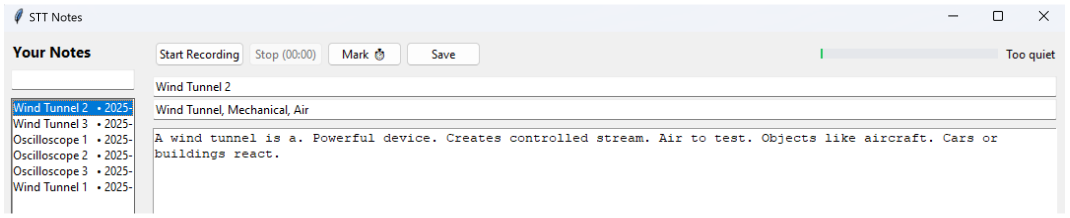
Wind Tunnel 1

Wind Tunnel, Mechanical, Air

```
Go to lexus. What channel. Power. What is. Cast. Please.
```

Analysis: Short chunks resulted in very fast feedback but low accuracy because linguistic context was reduced and words were sometimes split across chunk boundaries. This resulted in many incomplete words being transcribed to words that do not accurately represent the script.

Chunk Length = 1000 ms



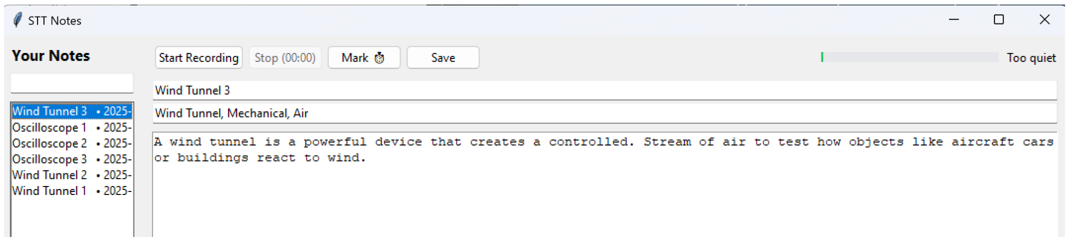
Wind Tunnel 2

Wind Tunnel, Mechanical, Air

```
A wind tunnel is a. Powerful device. Creates controlled stream. Air to test. Objects like aircraft. Cars or
buildings react.
```

Analysis: This length provided a better compromise between responsiveness and accuracy. Words were less likely to be fragmented, leading to improved stability. Words and sentences were still commonly cutoff, however, and this resulted in some inaccuracies in transcription.

Chunk Length = 2000 ms



Wind Tunnel 3

Wind Tunnel, Mechanical, Air

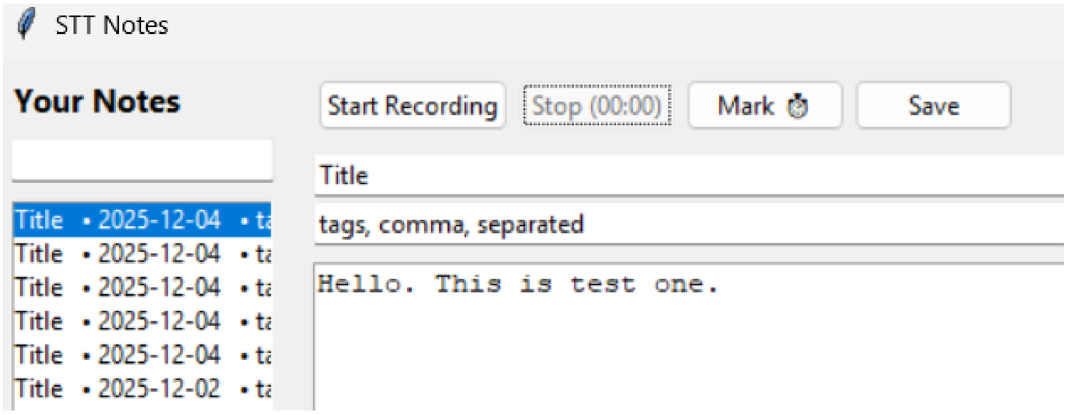
```
A wind tunnel is a powerful device that creates a controlled. Stream of air to test how objects like aircraft cars
or buildings react to wind.
```

Analysis: Larger chunks improved accuracy because the recognizer received full phrases, but the delay before text appeared increased noticeably. Most words were accurately transcribed with only minor mistakes.

5.3 Note Length (Short vs. Long Input)

Finally, we evaluated whether the system handles different lengths of spoken input differently.

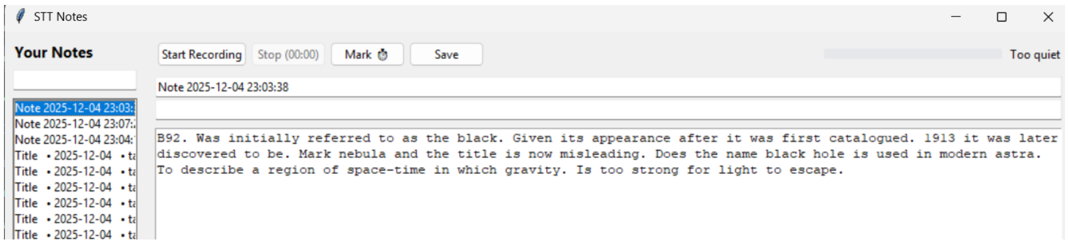
Short Script Test



```
Hello. This is test one.
```

Analysis: Short phrases were transcribed clearly and quickly. The system captured complete sentences with minimal delay.

Long Script Test



```
B92. Was initially referred to as the black. Given its appearance after it was first catalogued. 1913 it was later
discovered to be. Mark nebula and the title is now misleading. Does the name black hole is used in modern astra.
To describe a region of space-time in which gravity. Is too strong for light to escape.
```

Analysis: Long-form speech highlighted the impact of chunk length and noise settings. The transcript remained stable overall, but long pauses and breathing sometimes introduced fragmentations or repeated text. The long-form speech was still accurate to the point of being useful for use cases like lectures or meetings.

5.4 Summary of Findings

Across all tests, we found clear trade-offs between responsiveness, noise suppression, and transcription accuracy. Higher thresholds suppressed too much speech, while lower thresholds increased noise but preserved accuracy. Similarly, shorter chunks improved latency but reduced accuracy, while longer chunks provided superior recognition at the cost of increased delay. These results align with the expected behavior of chunk-based speech recognition pipelines.

6 What We Learned

One Python concept that was heavily applied in this project was object-oriented programming. Components such as the microphone input and user interface were implemented as classes (functions in the mic input and App classes use the instance method). Splitting these parts of the code into classes made it simple to write each part separately, then integrate them. For example, an instance of the MicrophoneInput class is created inside the App class, allowing functions within the mic input class such as start stream to be called within the App class. We also gained practice using Python libraries such as NumPy, Sounddevice, speech recognition, tkinter, etc. Throughout the coding process, we learned many technical skills such as how to process audio (using sounddevice library, deciding sample rate, chunk duration, etc.), how to convert speech to text (working with Google Web Speech API, testing various chunk durations, using threading, etc.), and how to create a graphical user interface (working with Tkinter, integrating backend logic using threading and callbacks, etc.). We also gained practice in dividing coding tasks, working separately, then collaborating in person and through GitHub commits in order to combine sections of the code.

7 Conclusion and Future Work

We were able to successfully implement a Python-based Speech-to-Text (STT) app that combined reliable real-time capture and chunking of audio, robust transcription with basic error handling and domain-specific normalization, and a Tkinter based graphical user interface (GUI). Through testing, we concluded that the app was able to reliably transcribe short phrases. However, when transcribing longer streams of audio, it would cut off the phrase at whatever the chunk length was set to in Settings (with a chunk length of 2000ms, it sent audio chunks to transcription every two seconds), leading to broken sentences at the points at which Google STT reset and began taking in a new chunk. One future improvement that would address this issue would be to use a recognizer such as Google Cloud Streaming API that can handle continuous audio, rather than Google Web Speech API, which does not register context between chunks. A streaming speech recognizer would also allow text to be output continuously, as speech is being recognized, rather than waiting until the entire chunk of audio is collected. Another feature we would consider adding is speaker diarization using Pyannote (finds points at which speaker changes, and groups segments of audio based on the speaker's characteristics). This would allow us to accurately transcribe conversations, which would be a useful tool for meetings, class discussions, etc.

References

[1] https://pypi.org/project/sounddevice/

[2] https://numpy.org/

[3] https://pypi.org/project/SpeechRecognition/

[4] https://docs.python.org/3/library/tkinter.html

A Appendix A: Source Code

```
# ===========================================
# STT Notes App: Mic + Transcription + Tk UI
# Cassidy Sakamoto, Nathan Tan, and Justin Glabicki
# ===========================================
#
# - Captures microphone audio in small chunks using sounddevice.
# - Streams audio into a simple transcription backend (Google Web API).
# - Normalizes text and sends completed segments into a Tkinter notes UI.
# - Notes (with tags, timestamps, and durations) are stored in local JSON files.

# ========================
# BACKEND: Mic + Transcription
# ========================

# sounddevice: live audio recording with NumPy arrays, supports callbacks
import sounddevice as sd

# numpy: basic numerical array operations, used for mic level and PCM conversion
import numpy as np

# Other imports for timing, byte packing, threading, and speech recognition
import time, struct, sys, threading
import speech_recognition as sr

# ======================================================
# Microphone input class: reads audio in short chunks
# ======================================================
class MicrophoneInput:
    """
␣␣␣␣Wraps␣a␣sounddevice␣InputStream␣to␣capture␣mono␣audio␣in␣small␣chunks.

␣␣␣␣-␣Calls␣audio_callback␣every␣time␣the␣sounddevice␣stream␣has␣new␣audio.
␣␣␣␣-␣Computes␣an␣RMS␣level␣for␣a␣simple␣mic␣level␣meter.
␣␣␣␣-␣Forwards␣the␣audio␣chunk␣to␣a␣user␣callback␣for␣transcription.
␣␣␣␣"""

    def __init__(self, threshold=0.0000001, sample_rate=48000, chunk_dur=0.02):
        """
␣␣␣␣␣␣␣␣Initialize␣microphone␣input␣parameters.

␣␣␣␣␣␣␣␣param␣threshold:␣Noise␣gate␣RMS␣threshold␣(float).
␣␣␣␣␣␣␣␣param␣sample_rate:␣Sampling␣rate␣in␣Hz.
␣␣␣␣␣␣␣␣param␣chunk_dur:␣Duration␣of␣each␣audio␣chunk␣in␣seconds.
␣␣␣␣␣␣␣␣"""
        self.threshold = threshold                      # RMS noise gate threshold
        self.quiet_frames = 0                           # counts consecutive quiet frames
        self.gate_limit = 5                             # number of consecutive quiet frames
             required to gate
        self.rate = sample_rate                         # Audio sample rate (Hz)
        self.chunk_dur = chunk_dur                      # Chunk duration in seconds
        self.chunk_size = int(self.rate * chunk_dur)    # Samples per chunk
        self.stream = None                              # sounddevice InputStream instance
        self.callback = None                            # Callback for processed audio
            chunks
        self.level_callback = None                      # Callback for mic level (RMS)
            updates

    def audio_callback(self, indata, frames, time_info, status):
        """
␣␣␣␣␣␣␣␣sounddevice␣callback.

␣␣␣␣␣␣␣␣-␣flattens␣input␣audio␣into␣a␣1D␣NumPy␣array,
␣␣␣␣␣␣␣␣-␣calculates␣RMS␣for␣a␣volume␣meter,
␣␣␣␣␣␣␣␣-␣optionally␣forwards␣RMS␣to␣a␣level_callback,
␣␣␣␣␣␣␣␣-␣forwards␣audio␣to␣the␣transcription␣callback␣if␣set.
␣␣␣␣␣␣␣␣"""
        # Print status if the stream reports any error or warning.
        if status:
            print(f"Stream␣status:␣{status}")

        # Convert audio buffer into a 1D NumPy array.
        audio_chunk = indata.flatten()

        # Compute RMS (root mean square) as a simple loudness estimate.
        rms = np.sqrt(np.mean(audio_chunk**2))

        # Console mic level meter for debugging.
        bar = "#" * int(rms * 50)
        print(f"Mic␣Level:␣{bar}␣({rms:.3f})")

        # Send current RMS level to the UI-level callback, if present.
        if self.level_callback:
            self.level_callback(float(rms))

       # Noise gate (Option 2A): require several consecutive quiet frames before gating
        if rms < self.threshold:
            self.quiet_frames += 1
        else:
            self.quiet_frames = 0  # reset on speech

        # If quiet for too long, do not forward audio for transcription
        if self.quiet_frames > self.gate_limit:
            return

        # Send audio chunk to the registered processing callback, if any.
        if self.callback:
            # The callback is expected to accept a NumPy array of float samples.
            self.callback(audio_chunk)

    def start_stream(self, callback=None, level_callback=None):
        """
␣␣␣␣␣␣␣␣Start␣the␣microphone␣stream␣and␣register␣callbacks.

␣␣␣␣␣␣␣␣:param␣callback:␣Function␣taking␣one␣NumPy␣array␣of␣audio␣samples␣(float32).
␣␣␣␣␣␣␣␣:param␣level_callback:␣Function␣taking␣a␣float␣RMS␣level␣for␣visual␣feedback.
␣␣␣␣␣␣␣␣"""
        # Save user callbacks for audio and level.
        self.callback = callback
        self.level_callback = level_callback

        # Create an input audio stream from the sounddevice library.
        self.stream = sd.InputStream(
            # device=9,                       # Input device ID (update as needed for your
                system)
            channels=1,                     # Mono input
            samplerate=self.rate,           # Sample rate in Hz
            callback=self.audio_callback  # Called whenever new audio arrives
        )

        # Start the audio stream.
        self.stream.start()
        print("Microphone␣stream␣started")

    def stop_stream(self):
        """
␣␣␣␣␣␣␣␣Stop␣and␣close␣the␣microphone␣stream␣if␣it␣is␣currently␣running.
␣␣␣␣␣␣␣␣"""
        if self.stream:
            self.stream.stop()
            self.stream.close()
            self.stream = None
            print("Microphone␣stream␣stopped")


# ==============================
# Transcription: framing + Google
# ==============================

# Audio framing constants
RATE = 48000
FRAME_MS = 20
SAMPLES_PER_FRAME = int(RATE * FRAME_MS / 1000)      # Samples per 20 ms
BYTES_PER_SAMPLE = 2

# Default chunk length (ms); will be overridden by config at runtime
CHUNK_MS = 6000                                      # ~6 s by default
CHUNK_FRAMES = int(CHUNK_MS / FRAME_MS)              # Number of 20 ms frames per chunk


# Transcription logic parameters
RETRY_LIMIT = 3
ENERGY_MIN = 200

# Domain-specific replacements and light punctuation handling
DOMAIN_REPLACE = {
    "r␣c": "RC",
    "p␣w␣m": "PWM",
    "k␣h␣z": "kHz",
    "h␣z": "Hz",
    "m␣s": "ms",
    "so␣c": "SoC",
}

# Global speech recognizer instance (Google Web API)
recognizer = sr.Recognizer()

# Global callback so the UI can receive finished transcript chunks.
transcript_callback = None

# ============================================================
# Helper functions for audio processing and recognition
# ============================================================

def frame_energy(frame_bytes: bytes) -> int:
    """
␣␣␣␣Compute␣average␣absolute␣sample␣value␣(a␣basic␣energy␣measure)
␣␣␣␣for␣a␣frame␣of␣16-bit␣PCM␣audio.

␣␣␣␣param␣frame_bytes:␣Raw␣PCM␣data.
␣␣␣␣return:␣Average␣magnitude␣across␣samples␣(integer).
␣␣␣␣"""
    n = len(frame_bytes) // 2
    if n == 0:
        return 0
    total = 0
    for i in range(0, len(frame_bytes), 2):
        total += abs(struct.unpack("<h", frame_bytes[i:i+2])[0])
    return total // n

def recognize_google_raw(raw_pcm: bytes) -> str:
    """
␣␣␣␣Send␣raw␣16-bit␣PCM␣audio␣to␣Google␣Web␣Speech␣API␣using
␣␣␣␣the␣speech_recognition␣library,␣and␣return␣the␣recognized␣text.
␣␣␣␣"""
    audio = sr.AudioData(raw_pcm, RATE, BYTES_PER_SAMPLE)
    return recognizer.recognize_google(audio)

def normalize_text(text: str) -> str:
    """
␣␣␣␣Normalize␣transcribed␣text:
␣␣␣␣-␣make␣lowercase␣for␣matching,
␣␣␣␣-␣apply␣domain-specific␣phrase␣replacements,
␣␣␣␣-␣collapse␣extra␣whitespace,
␣␣␣␣-␣ensure␣a␣simple␣trailing␣punctuation␣mark,
␣␣␣␣-␣then␣capitalize␣the␣first␣character.

␣␣␣␣param␣text:␣Raw␣text␣from␣recognizer.
␣␣␣␣return:␣Cleaned,␣nicely␣formatted␣sentence.
␣␣␣␣"""
    t = "␣" + text.lower() + "␣"
    for k, v in DOMAIN_REPLACE.items():
        t = t.replace(f"␣{k}␣", f"␣{v}␣")
    t = "␣".join(t.split())
    if t and t[-1] not in ".!?":
        t += "."
    return t[0].upper() + t[1:] if t else t

# ============================================================
# Integration between mic stream and transcription
# ============================================================

# Frame buffer for current chunk (~6 seconds of audio)
frames = []                 # Holds individual short frames (~20 ms each)
chunk_id = 0                # Simple counter for chunks, used for logging
partials = []               # Stores partial transcripts for debugging
t_chunk_start = time.time() # Start time for the current chunk (seconds)

def process_audio(chunk_np):
    """
␣␣␣␣Called␣by␣MicrophoneInput␣for␣each␣small␣audio␣chunk␣(NumPy␣array).

␣␣␣␣-␣Converts␣float␣samples␣(-1.0␣to␣1.0)␣to␣16-bit␣PCM␣bytes.
␣␣␣␣-␣Aggregates␣frames␣until␣a␣full␣chunk␣(about␣6␣seconds)␣is␣collected.
␣␣␣␣-␣When␣a␣chunk␣is␣ready,␣spawns␣a␣background␣thread␣to␣run␣transcription.
␣␣␣␣"""
    global frames, chunk_id, t_chunk_start

    # Convert float32 [-1.0, 1.0] to int16 PCM bytes.
    frame_bytes = (chunk_np * 32767).astype(np.int16).tobytes()
    frames.append(frame_bytes)

    # Once we have enough frames for a chunk, run transcription.
    if len(frames) >= CHUNK_FRAMES:
        raw = b"".join(frames)
        frames.clear()
        t_chunk_end = time.time()

        # Perform transcription on a background thread so we do not
        # block the real-time audio callback.
        threading.Thread(
            target=transcribe_chunk,
            args=(raw, chunk_id, t_chunk_start, t_chunk_end),
            daemon=True,
        ).start()

        chunk_id += 1
        t_chunk_start = time.time()

def transcribe_chunk(raw, chunk_id, t_start, t_end):
    """
␣␣␣␣Perform␣transcription␣on␣one␣audio␣chunk,␣with␣retry␣logic.

␣␣␣␣-␣Uses␣Google␣Web␣API␣via␣speech_recognition.
␣␣␣␣-␣On␣success,␣normalizes␣the␣text␣and␣appends␣to␣partials.
␣␣␣␣-␣If␣transcript_callback␣is␣set,␣forwards␣the␣normalized␣text␣to␣the␣UI.
␣␣␣␣-␣Logs␣warnings␣on␣failure␣or␣unknown␣speech.
␣␣␣␣"""
    global transcript_callback

    err = None
    text = ""
    for attempt in range(1, RETRY_LIMIT + 1):
        try:
            text = recognize_google_raw(raw)
            break
        except sr.UnknownValueError:
            # Speech recognizer could not understand this chunk.
            err = "speech␣not␣understood"
            print(f"Warning␣chunk␣{chunk_id}␣could␣not␣understand␣speech")
            break
        except sr.RequestError as e:
            # Network or API error; retry a few times if configured.
            err = f"api␣{e}"
            if attempt < RETRY_LIMIT:
                print(f"Warning␣chunk␣{chunk_id}␣retry␣{attempt}/{RETRY_LIMIT}")
                time.sleep(0.25)
            else:
                print(f"Warning␣chunk␣{chunk_id}␣failed␣after␣{RETRY_LIMIT}␣attempts:␣{e}")

    timing = f"[chunk␣{chunk_id}␣{t_start:.3f}-{t_end:.3f}]"
    if text:
        out = normalize_text(text)
        partials.append(out)
        print(f"{timing}␣{out}")

        # Send the finished text to the UI handler, if registered.
        if transcript_callback:
            try:
                transcript_callback(out)
            except Exception as e:
                print(f"Error␣in␣transcript_callback:␣{e}", file=sys.stderr)
    else:
        print(f"{timing}␣(no␣text,␣{err})")


# ========================
# UI + Notes App
# ========================


import json, uuid, random
from dataclasses import dataclass, asdict
from pathlib import Path
from typing import List, Callable, Optional
import tkinter as tk
from tkinter import ttk, filedialog, messagebox

# ---------------- Basics ----------------

# Directory for storing notes, config, and dictionary JSON files.
DATA_DIR = Path.cwd() / "stt_ui_data"
DATA_DIR.mkdir(exist_ok=True)
NOTES_FILE = DATA_DIR / "notes.json"
CONFIG_FILE = DATA_DIR / "config.json"
DICT_FILE = DATA_DIR / "dictionary.json"

def now_ms() -> int:
    """
␣␣␣␣Return␣the␣current␣time␣in␣milliseconds␣since␣the␣epoch.
␣␣␣␣"""
    return int(time.time() * 1000)

def format_time(ms: int) -> str:
    """
␣␣␣␣Convert␣a␣duration␣in␣milliseconds␣to␣a␣MM:SS␣string.

␣␣␣␣param␣ms:␣Duration␣in␣milliseconds.
␣␣␣␣return:␣String␣of␣the␣form␣"mm:ss".
␣␣␣␣"""
```

```
    s = ms // 1000
    m, sec = (s % 3600) // 60, s % 60
    return f"{m:02d}:{sec:02d}"

def slugify(name: str) -> str:
    """
␣␣␣␣Convert␣a␣note␣title␣into␣a␣simple␣filename-friendly␣slug.

␣␣␣␣-␣Keeps␣alphanumeric␣characters␣plus␣spaces,␣hyphens,␣and␣underscores.
␣␣␣␣-␣Replaces␣other␣characters␣with␣underscores.
␣␣␣␣-␣Collapses␣whitespace␣into␣single␣underscores.

␣␣␣␣param␣name:␣Original␣note␣title.
␣␣␣␣return:␣Sanitized␣slug␣string.
␣␣␣␣"""
    keep = "".join(c if (c.isalnum() or c in "␣-_") else "_" for c in name)
    return "_".join(keep.strip().split())

# ---------------- Data model ----------------

@dataclass
class Note:
    """
␣␣␣␣Represents␣a␣single␣note␣in␣the␣application.
␣␣␣␣"""
    id: str
    title: str
    content: str
    tags: List[str]
    createdAt: int
    updatedAt: int
    durationMs: int = 0

def load_notes() -> List[Note]:
    """
␣␣␣␣Load␣notes␣from␣NOTES_FILE,␣returning␣a␣list␣of␣Note␣instances.

␣␣␣␣If␣parsing␣fails␣or␣the␣file␣does␣not␣exist,␣an␣empty␣list␣is␣returned.
␣␣␣␣"""
    if NOTES_FILE.exists():
        try:
            raw = json.loads(NOTES_FILE.read_text(encoding="utf-8"))
            return [Note(**n) for n in raw]
        except Exception:
            # On error, return an empty list instead of crashing.
            pass
    return []

def save_notes(notes: List[Note]) -> None:
    """
␣␣␣␣Save␣the␣given␣list␣of␣notes␣to␣NOTES_FILE␣in␣JSON␣format.
␣␣␣␣"""
    NOTES_FILE.write_text(json.dumps([asdict(n) for n in notes], indent=2), encoding="utf-8"
        )

# --- config / dictionary helpers ---

def load_config() -> dict:
    """
␣␣␣␣Load␣configuration␣from␣CONFIG_FILE␣or␣return␣defaults␣if␣missing/invalid.
␣␣␣␣"""
    if CONFIG_FILE.exists():
        try:
            return json.loads(CONFIG_FILE.read_text(encoding="utf-8"))
        except Exception:
            pass
    # Default config values.
    return {
        "autosave_enabled": True,
        "autosave_delay_ms": 800,
        "noise_gate_threshold": 0.002,
        "chunk_ms": 2000,
        "max_retries": 3,
        "use_local_model": True,
    }

def save_config(cfg: dict) -> None:
    """
␣␣␣␣Persist␣the␣configuration␣dictionary␣to␣CONFIG_FILE.
␣␣␣␣"""
    CONFIG_FILE.write_text(json.dumps(cfg, indent=2), encoding="utf-8")

def load_dictionary() -> List[str]:
    """
␣␣␣␣Load␣a␣list␣of␣domain-specific␣words␣from␣DICT_FILE.

␣␣␣␣Returns␣an␣empty␣list␣if␣the␣file␣does␣not␣exist␣or␣parsing␣fails.
␣␣␣␣"""
    if DICT_FILE.exists():
        try:
            data = json.loads(DICT_FILE.read_text(encoding="utf-8"))
            if isinstance(data, list):
                return [str(w) for w in data]
        except Exception:
            pass
    return []

def save_dictionary(words: List[str]) -> None:
    """
␣␣␣␣Save␣a␣sorted,␣de-duplicated␣list␣of␣words␣to␣DICT_FILE.
␣␣␣␣"""
    DICT_FILE.write_text(json.dumps(sorted(set(words)), indent=2), encoding="utf-8")

# ---------------- Recorder adapter ----------------

class RecorderAdapter:
    """
␣␣␣␣Minimal␣interface␣that␣any␣recorder␣implementation␣should␣follow.

␣␣␣␣This␣allows␣the␣UI␣to␣work␣with␣either␣a␣real␣microphone␣backend
␣␣␣␣or␣a␣dummy␣recorder␣for␣design␣and␣testing.
␣␣␣␣"""
    def start(self, on_level: Callable[[float], None]) -> None: ...
    def stop(self) -> None: ...
    def is_running(self) -> bool: return False

class DummyRecorder(RecorderAdapter):
    """
␣␣␣␣UI␣development␣stub:␣fakes␣microphone␣levels␣so␣the␣level␣meter
␣␣␣␣could␣be␣designed␣and␣interaction␣flows␣without␣a␣real␣audio␣backend.
␣␣␣␣"""

    def __init__(self, root: tk.Tk):
        self.root = root
        self._running = False
        self._cb: Optional[Callable[[float], None]] = None

    def start(self, on_level: Callable[[float], None]) -> None:
        """
␣␣␣␣␣␣␣␣Start␣generating␣fake␣mic␣levels␣and␣call␣on_level␣periodically.
␣␣␣␣␣␣␣␣"""
        self._running, self._cb = True, on_level
        self._tick()

    def _tick(self):
        """
```

```
␣␣␣␣␣␣␣␣Internal␣periodic␣callback␣that␣updates␣the␣fake␣mic␣level
␣␣␣␣␣␣␣␣while␣the␣dummy␣recorder␣is␣running.
␣␣␣␣␣␣␣␣"""
        if not self._running: return
        # Simulate a range of levels from quiet to relatively loud.
        level = max(0.0, min(0.6, random.uniform(0.01, 0.5)))
        if self._cb: self._cb(level)
        self.root.after(80, self._tick)

    def stop(self) -> None:
        """
␣␣␣␣␣␣␣␣Stop␣generating␣fake␣microphone␣levels.
␣␣␣␣␣␣␣␣"""
        self._running = False
        self._cb = None

    def is_running(self) -> bool:
        """
␣␣␣␣␣␣␣␣Return␣True␣if␣the␣dummy␣recorder␣is␣currently␣generating␣levels.
␣␣␣␣␣␣␣␣"""
        return self._running


# ---------------- Main Tkinter App ----------------

class App(tk.Tk):
    """
␣␣␣␣Main␣Tkinter␣application␣for␣the␣STT␣Notes␣app.

␣␣␣␣-␣Manages␣notes␣list,␣search,␣and␣text␣editor.
␣␣␣␣-␣Starts/stops␣microphone␣recording␣and␣displays␣mic␣level.
␣␣␣␣-␣Receives␣transcript␣chunks␣and␣inserts␣them␣into␣the␣current␣note.
␣␣␣␣"""

    def __init__(self):
        super().__init__()
        self.title("STT␣Notes")
        self.geometry("1080x680")
        self.minsize(940, 580)

        # Notes data and selection state
        self.notes: List[Note] = load_notes()
        self.selected_id: Optional[str] = None

        # Config / dictionary
        self.config_data: dict = load_config()
        self.dictionary: List[str] = load_dictionary()

        # Apply chunk length from config to transcription backend
        global CHUNK_MS, CHUNK_FRAMES
        cfg_chunk = int(self.config_data.get("chunk_ms", 2000) or 2000)
        CHUNK_MS = max(500, cfg_chunk)          # clamp to at least 500 ms
        CHUNK_FRAMES = int(CHUNK_MS / FRAME_MS)

        global RETRY_LIMIT
        RETRY_LIMIT = int(self.config_data.get("max_retries", 3))

        # Autosave / edit tracking
        self.loading_note = False
        self.last_edit_ms = 0
        self.last_autosave_ms = 0

        # Recording state (timer and level)
        self.rec_start_ms = 0
        self.elapsed_ms = 0
        self.level_val = 0.0
        self.recording = False  # Tracks whether the mic stream is running
        # Real microphone input using MicrophoneInput backend
        noise_gate = float(self.config_data.get("noise_gate_threshold", 0.002) or 0.0)
        self.mic = MicrophoneInput(threshold=noise_gate, sample_rate=RATE)

        # UI state variables
        self.query_var = tk.StringVar()
        self.title_var = tk.StringVar()
        self.tags_var = tk.StringVar()

        # Build the UI layout and behaviors
        self._build()
        self._bind_shortcuts()
        self._refresh_list()

        # Connect backend transcription to this UI via a global callback
        global transcript_callback

        def ui_transcript_handler(text: str):
            """
␣␣␣␣␣␣␣␣␣␣␣␣Handle␣new␣transcript␣text␣arriving␣from␣the␣transcription␣backend.

␣␣␣␣␣␣␣␣␣␣␣␣Uses␣self.after␣to␣ensure␣updates␣happen␣on␣the␣Tk␣main␣thread.
␣␣␣␣␣␣␣␣␣␣␣␣"""
            self.after(0, lambda: self._append_transcript(text))

        transcript_callback = ui_transcript_handler

        # Start periodic UI updates (level meter, autosave, timer, etc.)
        self._ui_tick()

    # ----- layout -----
    def _build(self):
        """
␣␣␣␣␣␣␣␣Construct␣the␣full␣UI␣layout:␣sidebar,␣editor,␣controls,␣and␣footer.
␣␣␣␣␣␣␣␣"""
        self.columnconfigure(0, weight=0)
        self.columnconfigure(1, weight=1)
        self.rowconfigure(0, weight=1)

        # Sidebar: search and notes list
        side = ttk.Frame(self, padding=8)
        side.grid(row=0, column=0, sticky="nsew")
        side.rowconfigure(2, weight=1)

        ttk.Label(side, text="Your␣Notes", font=("Segoe␣UI", 11, "bold")).grid(row=0, column
            =0, sticky="w")
        ent = ttk.Entry(side, textvariable=self.query_var)
        self.search_entry = ent
        ent.grid(row=1, column=0, sticky="ew", pady=(6, 8))
        ent.bind("<KeyRelease>", lambda e: self._refresh_list())

        self.listbox = tk.Listbox(side, activestyle="dotbox")
        self.listbox.grid(row=2, column=0, sticky="nsew")
        self.listbox.bind("<<ListboxSelect>>", self._on_select)
        # Allow Delete key to delete the selected note.
        self.listbox.bind("<Delete>", lambda e: self._delete())

        # Delete button under the list
        btn_row = ttk.Frame(side)
        btn_row.grid(row=3, column=0, sticky="ew", pady=(6, 0))
        ttk.Button(btn_row, text="Delete␣Selected", command=self._delete).pack(side="left")

        # Main area: controls, title/tags, text editor, footer
        main = ttk.Frame(self, padding=10)
        main.grid(row=0, column=1, sticky="nsew")
        main.columnconfigure(0, weight=1)
        main.rowconfigure(5, weight=1)
        # Top control bar: record, stop, mark, save, mic level
        top = ttk.Frame(main)
        top.grid(row=0, column=0, sticky="ew", pady=(0, 8))
        top.columnconfigure(4, weight=1)

        self.start_btn = ttk.Button(top, text="Start␣Recording", command=self._start)
        self.start_btn.grid(row=0, column=0, padx=2)

        self.stop_btn = ttk.Button(top, text="Stop␣(00:00)", command=self._stop, state="
            disabled")
        self.stop_btn.grid(row=0, column=1, padx=2)

        ttk.Button(top, text="Mark␣   ", command=self._mark).grid(row=0, column=2, padx=2)
        ttk.Button(top, text="Save", command=self._save).grid(row=0, column=3, padx=2)

        # Level meter (visual mic level + text)
        meter = ttk.Frame(top)
        meter.grid(row=0, column=4, sticky="e")
        self.level_canvas = tk.Canvas(meter, width=180, height=10, bg="#E5E7EB",
            highlightthickness=0)
        self.level_canvas.pack(side="left", padx=(0, 6))
        self.level_bar = self.level_canvas.create_rectangle(0, 0, 0, 10, fill="#22C55E",
            width=0)
        self.level_label = ttk.Label(meter, text="Good")
        self.level_label.pack(side="left")

        # Title / tags entries
        self.title_entry = ttk.Entry(main, textvariable=self.title_var)
        self.title_entry.grid(row=1, column=0, sticky="ew", pady=(2, 2))
        self.title_entry.insert(0, "Title")

        self.tags_entry = ttk.Entry(main, textvariable=self.tags_var)
        self.tags_entry.grid(row=2, column=0, sticky="ew", pady=(0, 8))
        self.tags_entry.insert(0, "tags,␣comma,␣separated")

        # Track edits from title/tags for autosave.
        self.title_var.trace_add("write", self._on_content_edited)
        self.tags_var.trace_add("write", self._on_content_edited)

        # Main text editor for note content
        self.text = tk.Text(main, wrap="word")
        self.text.grid(row=5, column=0, sticky="nsew")
        self.text.bind("<<Modified>>", self._on_text_modified)

        # Footer row: status and auxiliary dialogs
        foot = ttk.Frame(main)
        foot.grid(row=6, column=0, sticky="ew", pady=8)

        # Status bar (right side)
        self.status_var = tk.StringVar(value="Ready")
        status_label = ttk.Label(foot, textvariable=self.status_var)
        status_label.pack(side="right")

        ttk.Button(foot, text="Settings", command=self._open_settings).pack(side="right",
            padx=(0, 8))
        ttk.Button(foot, text="Dictionary", command=self._open_dictionary).pack(side="right"
            , padx=(0, 8))

        # New note and export actions (left side)
        ttk.Button(foot, text="New␣Note", command=self._new_note).pack(side="left", padx=(0,
             8))
        ttk.Button(foot, text="Export␣.txt", command=lambda: self._export("txt")).pack(side=
            "left")
        ttk.Button(foot, text="Export␣.md", command=lambda: self._export("md")).pack(side="
            left", padx=(6, 0))

    # ----- keyboard shortcuts -----
    def _bind_shortcuts(self):
        """
␣␣␣␣␣␣␣␣Bind␣basic␣keyboard␣shortcuts␣for␣common␣actions.
␣␣␣␣␣␣␣␣"""
        self.bind_all("<Control-n>", lambda e: self._new_note())
        self.bind_all("<Control-s>", lambda e: self._save())
        self.bind_all("<Control-f>", lambda e: self._focus_search())
        self.bind_all("<Control-d>", lambda e: self._open_dictionary())
        self.bind_all("<Control-comma>", lambda e: self._open_settings())

    def _focus_search(self):
        """
␣␣␣␣␣␣␣␣Focus␣the␣search␣bar␣and␣select␣all␣text␣to␣make␣searching␣faster.
␣␣␣␣␣␣␣␣"""
        self.search_entry.focus_set()
        self.search_entry.select_range(0, "end")

    def _set_status(self, msg: str):
        """
␣␣␣␣␣␣␣␣Update␣the␣status␣bar␣text.
␣␣␣␣␣␣␣␣"""
        self.status_var.set(msg)

    # ----- recording controls (MicrophoneInput) -----
    def _start(self):
        """
␣␣␣␣␣␣␣␣Start␣microphone␣recording␣and␣update␣UI␣state.
␣␣␣␣␣␣␣␣"""
        if self.recording:
            return
        self.rec_start_ms = now_ms()
        self.elapsed_ms = 0

        # Connect mic stream to audio processing and mic level callbacks.
        self.mic.start_stream(callback=process_audio, level_callback=self._on_level)

        self.recording = True
        self.start_btn.configure(state="disabled")
        self.stop_btn.configure(state="normal")
        self._set_status("Recording...")

    def _stop(self):
        """
␣␣␣␣␣␣␣␣Stop␣microphone␣recording␣and␣finalize␣the␣recorded␣duration.
␣␣␣␣␣␣␣␣"""
        if not self.recording:
            return
        self.mic.stop_stream()
        self.elapsed_ms = now_ms() - self.rec_start_ms
        self.recording = False
        self.stop_btn.configure(text=f"Stop␣({format_time(0)})")
        self.start_btn.configure(state="normal")
        self.stop_btn.configure(state="disabled")
        self._set_status(f"Stopped.␣Duration␣{format_time(self.elapsed_ms)}")

    def _on_level(self, val: float):
        """
␣␣␣␣␣␣␣␣Receive␣mic␣RMS␣level␣from␣the␣backend␣thread␣and␣store␣it.

␣␣␣␣␣␣␣␣UI␣updates␣are␣performed␣in␣_ui_tick␣to␣stay␣on␣the␣Tk␣main␣thread.
␣␣␣␣␣␣␣␣"""
        self.level_val = max(0.0, float(val))

    def _mark(self):
        """
␣␣␣␣␣␣␣␣Insert␣a␣timestamp␣marker␣into␣the␣text␣at␣the␣current␣recording␣time.
␣␣␣␣␣␣␣␣"""
        elapsed = (now_ms() - self.rec_start_ms) if self.recording else self.elapsed_ms
        self.text.insert("end", f"\n[{format_time(elapsed)}]␣")
        self._on_content_edited()

    # ----- insert transcripts into the note -----
    def _append_transcript(self, text: str):
        """
␣␣␣␣␣␣␣␣Append␣recognized␣transcript␣text␣into␣the␣editor,␣with␣a␣space␣if␣needed.
␣␣␣␣␣␣␣␣"""
        existing = self.text.get("1.0", "end").strip()
        if existing:
            self.text.insert("end", "␣")
        self.text.insert("end", text)
        self._on_content_edited()

    # ----- notes list / search -----
    def _filtered(self) -> List[Note]:
        """
␣␣␣␣␣␣␣␣Return␣notes␣filtered␣by␣the␣current␣search␣query,␣sorted␣by␣updated␣time.
␣␣␣␣␣␣␣␣"""
        q = self.query_var.get().strip().lower()
        notes = sorted(self.notes, key=lambda n: n.updatedAt, reverse=True)
        if not q:
            return notes
        out = []
        for n in notes:
            if (q in n.title.lower()
                or q in n.content.lower()
                or any(q in t.lower() for t in n.tags)):
                out.append(n)
        return out

    def _refresh_list(self):
        """
␣␣␣␣␣␣␣␣Refresh␣the␣Listbox␣contents␣based␣on␣the␣current␣filtered␣notes.

␣␣␣␣␣␣␣␣Keeps␣the␣current␣selection␣(self.selected_id)␣whenever␣possible␣so
␣␣␣␣␣␣␣␣autosave␣or␣other␣refreshes␣do␣not␣"lose"␣which␣note␣is␣active.
␣␣␣␣␣␣␣␣"""
        items = self._filtered()

        # Rebuild the listbox entries.
        self.listbox.delete(0, "end")
        for n in items:
            date = time.strftime("%Y-%m-%d", time.localtime(n.createdAt / 1000))
            tags = ",␣".join(n.tags) if n.tags else ""
            self.listbox.insert("end", f"{n.title}␣␣␣   ␣{date}␣␣␣   ␣{tags}")

        # Try to restore selection based on selected_id.
        if self.selected_id is not None:
            for idx, n in enumerate(items):
                if n.id == self.selected_id:
                    self.listbox.selection_set(idx)
                    self.listbox.activate(idx)
                    self.listbox.see(idx)
                    break

    def _selected(self) -> Optional[Note]:
        """
␣␣␣␣␣␣␣␣Return␣the␣note␣currently␣selected␣in␣the␣Listbox.
␣␣␣␣␣␣␣␣"""
        sel = self.listbox.curselection()
        if not sel:
            return None
        idx = sel[0]
        items = self._filtered()
        if 0 <= idx < len(items):
            return items[idx]
        return None
    def _on_select(self, _evt=None):
        """
␣␣␣␣␣␣␣␣Handle␣Listbox␣selection␣changes␣and␣load␣the␣selected␣note.
␣␣␣␣␣␣␣␣"""
        n = self._selected()
        if not n:
            return
        self.loading_note = True
        try:
            self.selected_id = n.id
            self.title_var.set(n.title)
            self.tags_var.set(",␣".join(n.tags))
            self.text.delete("1.0", "end")
            self.text.insert("1.0", n.content)
            self.last_edit_ms = 0
        finally:
            self.loading_note = False

    # ----- edit tracking -----
    def _on_text_modified(self, _evt=None):
        """
␣␣␣␣␣␣␣␣Track␣edits␣from␣the␣main␣text␣widget␣for␣autosave.
␣␣␣␣␣␣␣␣"""
        if self.loading_note:
            self.text.edit_modified(False)
            return
        self.last_edit_ms = now_ms()
        self.text.edit_modified(False)

    def _on_content_edited(self, *args):
        """
␣␣␣␣␣␣␣␣Track␣edits␣from␣title/tags␣or␣other␣content␣sources␣for␣autosave.
␣␣␣␣␣␣␣␣"""
        if self.loading_note:
            return
        self.last_edit_ms = now_ms()

    def _has_meaningful_content(self) -> bool:
        """
␣␣␣␣␣␣␣␣Check␣whether␣the␣current␣note␣has␣any␣non-empty␣title,␣tags,␣or␣text.
␣␣␣␣␣␣␣␣"""
        if self.text.get("1.0", "end").strip():
            return True
        if self.title_var.get().strip():
            return True
        if self.tags_var.get().strip():
            return True
        return False

    # ----- new note / delete -----
    def _new_note(self):
        """
␣␣␣␣␣␣␣␣Prepare␣the␣UI␣for␣creating␣a␣brand-new␣note.
␣␣␣␣␣␣␣␣"""
        self.loading_note = True
        try:
            self.selected_id = None  # Ensures _save will create a new note
            self.title_var.set("")
            self.tags_var.set("")
            self.text.delete("1.0", "end")
            # Reset elapsed recording duration for this note.
            self.elapsed_ms = 0
            self.last_edit_ms = 0
        finally:
            self.loading_note = False
        self._set_status("New␣note")

    def _delete(self):
        """
␣␣␣␣␣␣␣␣Delete␣the␣currently␣selected␣note␣after␣confirming␣with␣the␣user.
␣␣␣␣␣␣␣␣"""
        n = self._selected()
        if not n:
            messagebox.showinfo("Delete", "Select␣a␣note␣to␣delete.")
            return

        if not messagebox.askyesno("Delete␣note",
                                   f"Delete␣'{n.title}'?\nThis␣cannot␣be␣undone."):
            return

        # Remove note from the list and save.
        self.notes = [note for note in self.notes if note.id != n.id]
        save_notes(self.notes)

        # Clear selection and editor.
        self.loading_note = True
        try:
            self.selected_id = None
            self.title_var.set("")
            self.tags_var.set("")
            self.text.delete("1.0", "end")
        finally:
            self.loading_note = False

        self._refresh_list()
        self._set_status("Note␣deleted.")

    # ----- save / export -----
    def _save(self, *, via_autosave: bool = False, show_popup: bool = True):
        """
␣␣␣␣␣␣␣␣Save␣the␣current␣note.␣If␣via_autosave␣is␣True,␣this␣is␣an␣autosave.

␣␣␣␣␣␣␣␣:param␣via_autosave:␣Set␣to␣True␣when␣called␣from␣the␣autosave␣logic.
␣␣␣␣␣␣␣␣:param␣show_popup:␣Show␣a␣confirmation␣popup␣if␣this␣is␣a␣manual␣save.
␣␣␣␣␣␣␣␣"""
        if not self._has_meaningful_content() and via_autosave:
            # Do not autosave completely empty notes.
            return

        title = (self.title_var.get() or "").strip() or f"Note␣{time.strftime('%Y-%m-%d␣%H:%
            M:%S')}"
        tags = [t.strip() for t in self.tags_var.get().split(",") if t.strip()]
        content = self.text.get("1.0", "end").rstrip()
        now_ts = now_ms()

        if self.selected_id:
            # Update existing note.
            for i, n in enumerate(self.notes):
                if n.id == self.selected_id:
                    self.notes[i] = Note(
                        id=n.id,
                        title=title,
                        content=content,
                        tags=tags,
                        createdAt=n.createdAt,
                        updatedAt=now_ts,
                        durationMs=n.durationMs or self.elapsed_ms,
                    )
                    break
        else:
            # Create a new note.
            nid = str(uuid.uuid4())
            self.notes.insert(0, Note(
                id=nid,
                title=title,
                content=content,
                tags=tags,
                createdAt=now_ts,
                updatedAt=now_ts,
                durationMs=self.elapsed_ms,
            ))
            self.selected_id = nid

        save_notes(self.notes)

        # Reset edit tracking.
        self.last_edit_ms = 0
        self.last_autosave_ms = now_ts

        if show_popup and not via_autosave:
            messagebox.showinfo("Saved", "Note␣saved.")

        timestamp = time.strftime("%H:%M:%S")
        if via_autosave:
            self._set_status(f"Autosaved␣at␣{timestamp}")
        else:
            self._set_status(f"Saved␣at␣{timestamp}")

        self._refresh_list()

    def _export(self, kind: str):
        """
␣␣␣␣␣␣␣␣Export␣the␣currently␣selected␣note␣as␣a␣.txt␣or␣.md␣file.

␣␣␣␣␣␣␣␣param␣kind:␣File␣extension␣without␣dot␣(e.g.,␣"txt"␣or␣"md").
␣␣␣␣␣␣␣␣"""
        n = self._selected()
        if not n:
            messagebox.showinfo("Export", "Select␣a␣note␣first.")
            return
        default = f"{slugify(n.title)}.{kind}"
        path = filedialog.asksaveasfilename(defaultextension=f".{kind}", initialfile=default
            )
        if not path:
            return
        Path(path).write_text(n.content, encoding="utf-8")
        messagebox.showinfo("Export", f"Exported␣to␣{path}")
        self._set_status("Exported␣note")

    # ----- Settings / Dictionary -----
    def _open_settings(self):
        """
␣␣␣␣␣␣␣␣Open␣(or␣bring␣to␣front)␣the␣settings␣window␣for␣autosave␣and␣backend␣parameters.
␣␣␣␣␣␣␣␣"""
        if hasattr(self, "_settings_win") and self._settings_win.winfo_exists():
            self._settings_win.lift()
            return

        win = tk.Toplevel(self)
        self._settings_win = win
        win.title("Settings")
        win.resizable(False, False)

        autosave_var = tk.BooleanVar(value=self.config_data.get("autosave_enabled", True))
        delay_var = tk.IntVar(value=self.config_data.get("autosave_delay_ms", 800))
        noise_var = tk.DoubleVar(value=self.config_data.get("noise_gate_threshold", 0.002))
        chunk_var = tk.IntVar(value=self.config_data.get("chunk_ms", 500))
        retries_var = tk.IntVar(value=self.config_data.get("max_retries", 3))
        local_var = tk.BooleanVar(value=self.config_data.get("use_local_model", True))

        row = 0
        ttk.Checkbutton(win, text="Enable␣autosave", variable=autosave_var).grid(
            row=row, column=0, columnspan=2, sticky="w", padx=8, pady=(8, 4)
        )
        row += 1
        ttk.Label(win, text="Autosave␣delay␣(ms):").grid(row=row, column=0, sticky="e", padx
            =8, pady=4)
        ttk.Entry(win, textvariable=delay_var, width=8).grid(row=row, column=1, sticky="w",
            padx=8, pady=4)
        row += 1

        # placeholders for backend team
        ttk.Label(win, text="Noise␣gate␣threshold:").grid(row=row, column=0, sticky="e",
            padx=8, pady=4)
        ttk.Entry(win, textvariable=noise_var, width=8).grid(row=row, column=1, sticky="w",
            padx=8, pady=4)
        row += 1

        ttk.Label(win, text="Chunk␣length␣(ms):").grid(row=row, column=0, sticky="e", padx
            =8, pady=4)
        ttk.Entry(win, textvariable=chunk_var, width=8).grid(row=row, column=1, sticky="w",
            padx=8, pady=4)
        row += 1

        ttk.Label(win, text="Max␣retries␣per␣chunk:").grid(row=row, column=0, sticky="e",
            padx=8, pady=4)
        ttk.Entry(win, textvariable=retries_var, width=8).grid(row=row, column=1, sticky="w"
            , padx=8, pady=4)
        row += 1

        ttk.Checkbutton(
            win, text="Prefer␣local␣model␣when␣available", variable=local_var
        ).grid(row=row, column=0, columnspan=2, sticky="w", padx=8, pady=4)
        row += 1

        def save_and_close():
            """
␣␣␣␣␣␣␣␣␣␣␣␣Save␣settings␣from␣the␣dialog␣back␣into␣config_data␣and␣close␣the␣window.
␣␣␣␣␣␣␣␣␣␣␣␣"""
            self.config_data["autosave_enabled"] = bool(autosave_var.get())
            self.config_data["autosave_delay_ms"] = max(100, int(delay_var.get() or 0))
            self.config_data["noise_gate_threshold"] = float(noise_var.get() or 0.0)
            self.config_data["chunk_ms"] = max(100, int(chunk_var.get() or 0))
            self.config_data["max_retries"] = max(0, int(retries_var.get() or 0))
            self.config_data["use_local_model"] = bool(local_var.get())

            global RETRY_LIMIT
            RETRY_LIMIT = int(self.config_data["max_retries"])

            # Apply updated settings to backend
            global CHUNK_MS, CHUNK_FRAMES
            CHUNK_MS = max(500, int(self.config_data["chunk_ms"]))
            CHUNK_FRAMES = int(CHUNK_MS / FRAME_MS)

            # Update mic noise gate threshold live (no restart needed)
            self.mic.threshold = float(self.config_data["noise_gate_threshold"])

            save_config(self.config_data)
            self._set_status("Settings␣saved")
            win.destroy()

        ttk.Button(win, text="Save", command=save_and_close).grid(
            row=row, column=0, columnspan=2, pady=(8, 8)
        )

    def _open_dictionary(self):
        """
␣␣␣␣␣␣␣␣Open␣(or␣bring␣to␣front)␣the␣domain␣dictionary␣editor␣window.
␣␣␣␣␣␣␣␣"""
        if hasattr(self, "_dict_win") and self._dict_win.winfo_exists():
            self._dict_win.lift()
            return
        win = tk.Toplevel(self)
        self._dict_win = win
        win.title("Domain␣Dictionary")
        win.resizable(False, False)

        frame = ttk.Frame(win, padding=8)
        frame.grid(row=0, column=0, sticky="nsew")

        ttk.Label(frame, text="Domain-specific␣words␣(one␣per␣entry):").grid(
            row=0, column=0, columnspan=2, sticky="w", pady=(0, 4)
        )

        listbox = tk.Listbox(frame, height=10)
        listbox.grid(row=1, column=0, columnspan=2, sticky="nsew")
        for w in self.dictionary:
            listbox.insert("end", w)

        entry_var = tk.StringVar()
        entry = ttk.Entry(frame, textvariable=entry_var)
        entry.grid(row=2, column=0, sticky="ew", pady=(4, 4))
        frame.columnconfigure(0, weight=1)

        def add_word():
            """
␣␣␣␣␣␣␣␣␣␣␣␣Add␣a␣new␣word␣from␣the␣entry␣field␣into␣the␣dictionary␣and␣listbox.
␣␣␣␣␣␣␣␣␣␣␣␣"""
            w = entry_var.get().strip()
            if not w:
                return
            if w not in self.dictionary:
                self.dictionary.append(w)
                listbox.insert("end", w)
                save_dictionary(self.dictionary)
                self._set_status("Dictionary␣updated")
            entry_var.set("")

        def delete_selected():
            """
␣␣␣␣␣␣␣␣␣␣␣␣Delete␣the␣currently␣selected␣word␣from␣the␣dictionary␣and␣listbox.
␣␣␣␣␣␣␣␣␣␣␣␣"""
            sel = listbox.curselection()
            if not sel:
                return
            idx = sel[0]
            word = listbox.get(idx)
            self.dictionary = [w for w in self.dictionary if w != word]
            listbox.delete(idx)
            save_dictionary(self.dictionary)
            self._set_status("Dictionary␣updated")

        ttk.Button(frame, text="Add", command=add_word).grid(
            row=2, column=1, sticky="w", padx=(4, 0)
        )
        ttk.Button(frame, text="Delete␣Selected", command=delete_selected).grid(
            row=3, column=0, columnspan=2, sticky="w", pady=(4, 0)
        )

    # ----- periodic UI updates -----
    def _ui_tick(self):
        """
␣␣␣␣␣␣␣␣Periodic␣UI␣ticker:

␣␣␣␣␣␣␣␣-␣Updates␣the␣mic␣level␣bar␣and␣label.
␣␣␣␣␣␣␣␣-␣Updates␣elapsed␣time␣on␣the␣Stop␣button␣while␣recording.
␣␣␣␣␣␣␣␣-␣Triggers␣autosave␣when␣conditions␣are␣met.
␣␣␣␣␣␣␣␣"""
        # Update meter bar and label based on current level_val.
        pct = max(0.0, min(1.0, self.level_val * 1.8))
        self.level_canvas.coords(self.level_bar, 0, 0, int(180 * pct), 10)
        self.level_label.configure(
            text="Too␣quiet" if self.level_val < 0.03 else ("Too␣loud" if self.level_val >
                0.35 else "Good")
        )

        # Update elapsed time on the Stop button while recording.
        if self.recording:
            el = now_ms() - self.rec_start_ms
            self.stop_btn.configure(text=f"Stop␣({format_time(el)})")

        # Autosave check.
        now_ts = now_ms()
        if (
            self.config_data.get("autosave_enabled", True)
            and self.last_edit_ms
            and self.last_edit_ms > self.last_autosave_ms
            and now_ts - self.last_edit_ms >= self.config_data.get("autosave_delay_ms", 800)
        ):
            self._save(via_autosave=True, show_popup=False)

        # Schedule the next tick.
        self.after(100, self._ui_tick)


if __name__ == "__main__":
    App().mainloop()
```