

Taurscribe Architecture Guide for Beginners

Perfect for: Complete beginners to programming, Rust newcomers, or anyone curious about how speech recognition works!

Goal: Understand how Taurscribe works through simple explanations, fun analogies, and visual diagrams.

Table of Contents

1. [What is Taurscribe?](#)
2. [The Big Picture](#)
3.  [Platform Support & Hardware Acceleration ← NEW!](#)
4.  [Audio Processing: Whisper vs Parakeet](#)
5. [Complete Audio Processing Flow](#)
6. [Rust Basics You Need to Know](#)
7. [Complete Flow: Start to Finish](#)
8. [Component Deep Dive](#)
9. [Understanding Rust Ownership](#)
10. [Dependencies Explained](#)
11. [Common Beginner Questions](#)
12. [Cumulative Context Feature](#)
13. [Annotated Rust Code Examples](#)
14. [Model Selection Feature](#)
15. [Voice Activity Detection \(VAD\)](#)
16.  [Module Architecture](#)
17. [File & Function Reference](#)

What is Taurscribe?

Taurscribe is a **desktop application** that listens to your voice and magically turns it into text using artificial intelligence!

Think of it like having a super-fast personal assistant that writes down everything you say.

Technology Stack (in plain English):

- **Frontend:** React + TypeScript (the pretty buttons and screens you see)
- **Backend:** Rust + Tauri (the super-fast engine that does all the hard work)
- **AI Engines:** Two powerful brains to choose from:
 -  **Whisper AI** - Very accurate, great for all situations
 -  **Parakeet Nemotron** - Lightning fast, optimized for real-time streaming

Key Features:

- Real-time transcription while you speak (see words appear as you talk!)
- High-quality final transcript when you stop

- GPU acceleration for blazing speed (uses your graphics card!)
 - Two AI engines to choose from (Whisper or Parakeet)
 - Multiple models for each engine (pick small & fast or large & accurate)
 - Voice Activity Detection (automatically skips silence)
-

⚡ Refactored Architecture (January 2026)

Taurscribe recently underwent a major code reorganization for better maintainability:

Before Refactoring

- ✗ lib.rs: 1,151 lines (monolithic, hard to navigate)
- ✗ Everything in one file
- ✗ Hard to find specific features

After Refactoring

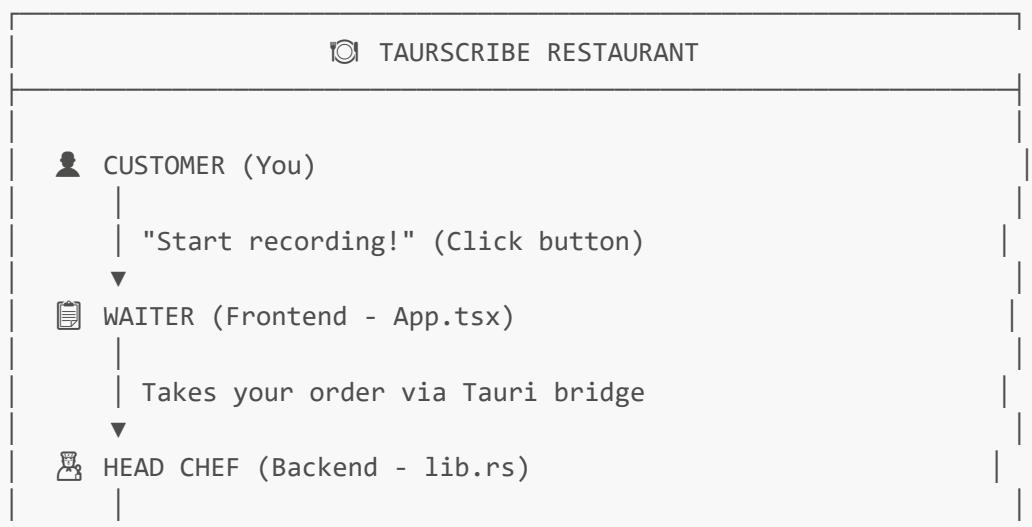
- lib.rs: 115 lines (90% reduction!)
- 19 focused modules organized by feature
- Easy to navigate: recording → commands/recording.rs
- Ready for team development

No Breaking Changes! The app works exactly the same - we just made the code much easier to maintain. See [Module Architecture](#) for details.

The Big Picture

⌚ Restaurant Analogy

Imagine Taurscribe as a **restaurant kitchen**. Here's how the pieces work together:



```
Organizes the kitchen

→ ⌚ SUPPLIER (Microphone)
    Fresh ingredients arrive every 10ms!

→ 🍳 COOK #1 (File Writer Thread)
    "I'll preserve the ingredients"
    Saves audio to WAV file

→ 🍳 COOK #2 (Whisper Thread)
    "I'll taste-test every 6 seconds"
    Real-time transcription preview

→ 🍳 MASTER CHEF (WhisperManager)
    "I analyze with AI precision"
    Final professional-quality transcription
```

⌚ Simple Data Flow

```
⌚ Your Voice

→ Microphone captures sound waves

→ Converts to numbers (samples)

→ Split into two streams:

→ Stream 1 → 📁 Save to disk (WAV file)

→ Stream 2 → 🎵 AI transcription → 📄 Text
```

💻 Platform Support & Hardware Acceleration

Taurscribe is designed to run on **multiple operating systems** and **leverage hardware acceleration** for maximum performance. This section explains how the application adapts to different platforms and hardware configurations.

🌐 Supported Platforms

Taurscribe supports the following platforms with optimized builds for each:

TAURSCRIBE PLATFORM MATRIX

- | | |
|---|--------------------------------------|
| <input checked="" type="checkbox"/> Windows (x86_64) | - CUDA + Vulkan + DirectML |
| <input checked="" type="checkbox"/> Windows (ARM64) | - DirectML + CPU (Snapdragon X) |
| <input checked="" type="checkbox"/> macOS (Apple Silicon) | - CoreML + Metal |
| <input checked="" type="checkbox"/> macOS (Intel) | - CPU + XNNPACK |
| <input checked="" type="checkbox"/> Linux (x86_64) | - CUDA + Vulkan + TensorRT |
| <input checked="" type="checkbox"/> Linux (ARM64) | - CPU + XNNPACK (Raspberry Pi, etc.) |

⚡ Hardware Acceleration Technologies

Taurscribe intelligently selects the best acceleration technology based on your hardware:

_CUDA (NVIDIA GPUs)

Platforms: Windows x64, Linux x64

Use Case: NVIDIA graphics cards (GeForce RTX, Quadro, Tesla)

Performance: ★★★★☆ **Best Performance** (up to 60x realtime)

```
// Enabled in Cargo.toml for Windows/Linux x86_64:
whisper-rs = { features = ["cuda", "vulkan"] }
parakeet-rs = { features = ["cuda"] }
ort = { features = ["cuda", "tensorrt"] }
```

How It Works:

- Whisper: CUDA kernel acceleration for encoder/decoder
- Parakeet: CUDA execution provider for ONNX Runtime
- Automatic fallback to CPU if NVIDIA driver not found

Detection Logic (`whisper.rs` lines 317-322):

```
fn is_cuda_available(&self) -> bool {
    std::process::Command::new("nvidia-smi")
        .output()
        .map(|output| output.status.success())
        .unwrap_or(false)
}
```

渲 Vulkan (Universal GPU)

Platforms: Windows x64, Linux x64

Use Case: AMD GPUs, Intel Arc, older NVIDIA GPUs

Performance: ★★★★☆ **Excellent** (up to 30x realtime)

```
// Enabled in Cargo.toml for Windows/Linux x86_64:
whisper-rs = { features = ["vulkan"] }
```

How It Works:

- Cross-platform GPU API (works on AMD, Intel, NVIDIA)
- Whisper: Vulkan compute shaders for parallel processing
- Automatic selection if CUDA not available

Use Cases:

- AMD Radeon RX 6000/7000 series
- Intel Arc A-series
- NVIDIA GPUs without CUDA drivers

DirectML (Windows Universal GPU)

Platforms: Windows x64, Windows ARM64

Use Case: Windows devices with any GPU or NPU (Neural Processing Unit)

Performance: ★ ★ ★ ★ **Very Good** (up to 25x realtime)

```
// Enabled in Cargo.toml for Windows:
[target.'cfg(all(target_os = "windows", target_arch = "x86_64"))'.dependencies]
parakeet-rs = { features = ["directml"] }
ort = { features = ["directml"] }

[target.'cfg(all(target_os = "windows", target_arch = "aarch64"))'.dependencies]
parakeet-rs = { features = ["directml"] }
ort = { features = ["directml"] }
```

How It Works:

- Microsoft's hardware-accelerated ML framework
- Works with ANY GPU (NVIDIA, AMD, Intel) via DirectX 12
- **Special:** Supports NPUs on Snapdragon X ARM processors
- Parakeet: DirectML execution provider for ONNX Runtime

Use Cases:

- Qualcomm Snapdragon X Elite/Plus (Windows ARM laptops)
- AMD Radeon GPUs on Windows
- Intel Iris Xe integrated graphics
- Surface devices with NPUs

CoreML (Apple Silicon)

Platforms: macOS ARM64 (M1/M2/M3/M4)

Use Case: Apple Silicon Macs

Performance: ★★☆☆★ **Excellent** (Neural Engine acceleration)

```
// Enabled in Cargo.toml for macOS:  
[target.'cfg(target_os = "macos")'.dependencies]  
ort = { features = ["coreml", "xnnpack"] }
```

How It Works:

- Parakeet: CoreML execution provider uses Apple Neural Engine
- Leverages dedicated ML hardware on Apple Silicon chips
- Fallback to Metal GPU if Neural Engine unavailable

Special Notes:

- Requires macOS 13.4+ (set in `build.rs`)
- Only applies to Parakeet (ONNX Runtime models)
- Whisper uses Metal GPU acceleration by default

TensorRT (NVIDIA Optimized Inference)

Platforms: Windows x64, Linux x64

Use Case: NVIDIA GPUs with TensorRT runtime installed

Performance: ★★☆☆★ **Maximum Speed** (up to 80x realtime)

```
// Enabled in Cargo.toml for Windows/Linux x86_64:  
ort = { features = ["tensorrt"] }
```

How It Works:

- NVIDIA's optimized inference engine
- Parakeet: TensorRT execution provider for ONNX models
- Requires separate TensorRT installation
- Automatic fallback to CUDA if TensorRT unavailable

XNNPACK (Optimized CPU)

Platforms: All (macOS, Windows, Linux, ARM)

Use Case: CPU fallback or devices without GPU

Performance: ★★☆☆★ **Good** (up to 10x realtime)

```
// Enabled in Cargo.toml for all platforms:  
ort = { features = ["xnnpack"] }
```

How It Works:

- Optimized CPU inference using SIMD instructions (SSE, AVX, NEON)
- Parakeet: XNNPACK execution provider for ONNX Runtime
- Automatic selection if no GPU acceleration available

Use Cases:

- Intel Macs (no CoreML support)
- Raspberry Pi / ARM Linux devices
- Virtual machines without GPU passthrough
- Fallback when GPU drivers missing

Pure CPU (Fallback)

Platforms: All

Use Case: No hardware acceleration available

Performance: ★★ Acceptable (2-5x realtime)

How It Works:

- Whisper: Multi-threaded CPU inference
- Parakeet: CPU execution provider for ONNX Runtime
- Always available as last resort

Platform-Specific Configuration

Cargo.toml Platform Targets

Tauri uses Rust's conditional compilation to include only the necessary features for each platform:

```
# ===== BASE DEPENDENCIES (All Platforms) =====
[dependencies]
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
parakeet-rs = { version = "=0.3.0" }
ort = { version = "2.0.0-rc.11", features = ["download-binaries"] }

# ===== PLATFORM-SPECIFIC OVERRIDES =====

# --- macOS (ARM & Intel) ---
[target.'cfg(target_os = "macos")'.dependencies]
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
ort = { features = ["download-binaries", "coreml", "xnnpack"] }
# CoreML: Apple Neural Engine (M1/M2/M3/M4)
# XNNPACK: CPU optimization (Intel Macs fallback)

# --- Windows (x86_64) ---
[target.'cfg(all(target_os = "windows", target_arch = "x86_64"))'.dependencies]
```

```

whisper-rs = { features = ["cuda", "vulkan"] }
parakeet-rs = { features = ["cuda", "directml"] }
ort = { features = ["cuda", "directml", "tensorrt", "xnnpack"] }
# CUDA: NVIDIA GPU (primary)
# DirectML: Universal Windows GPU (AMD/Intel fallback)
# TensorRT: NVIDIA optimized inference
# Vulkan: Whisper GPU acceleration
# XNNPACK: CPU optimization

# --- Linux (x86_64) ---
[target.'cfg(all(target_os = "linux", target_arch = "x86_64"))'.dependencies]
whisper-rs = { features = ["cuda", "vulkan"] }
parakeet-rs = { features = ["cuda"] }
ort = { features = ["cuda", "tensorrt", "xnnpack"] }
# CUDA: NVIDIA GPU (primary)
# Vulkan: Whisper GPU acceleration (AMD/Intel)
# TensorRT: NVIDIA optimized inference
# XNNPACK: CPU optimization

# --- Windows (ARM64) ---
[target.'cfg(all(target_os = "windows", target_arch = "aarch64"))'.dependencies]
parakeet-rs = { features = ["directml"] }
ort = { features = ["directml", "xnnpack"] }
# DirectML: Snapdragon X NPU/GPU acceleration
# XNNPACK: CPU optimization
# Note: No whisper-rs CUDA/Vulkan (ARM Windows limitation)

# --- Linux (ARM64) ---
[target.'cfg(all(target_os = "linux", target_arch = "aarch64"))'.dependencies]
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
parakeet-rs = { version = "=0.3.0" }
ort = { features = ["xnnpack"] }
# XNNPACK: CPU optimization for ARM (Raspberry Pi, etc.)
# No GPU acceleration (driver limitations on ARM Linux)

```

🔧 Build Configuration ([build.rs](#))

The [build.rs](#) file contains platform-specific build logic:

1. macOS Deployment Target

```

#[cfg(target_os = "macos")]
{
    // ONNX Runtime requires macOS 13.4+ for CoreML support
    println!("cargo:rustc-env=MACOSX_DEPLOYMENT_TARGET=13.4");
    std::env::set_var("MACOSX_DEPLOYMENT_TARGET", "13.4");
    std::env::set_var("CMAKE_OSX_DEPLOYMENT_TARGET", "13.4");
}

```

Why? CoreML and ONNX Runtime require modern macOS versions.

2. Windows ARM64 Clang Requirement

```
#[cfg(all(target_os = "windows", target_arch = "aarch64"))]
{
    // whisper.cpp requires Clang for ARM64 Windows (MSVC not supported)
    println!("cargo:warning=Building for Windows ARM64 - Clang/LLVM required");
    std::env::set_var("CC", "clang-cl");
    std::env::set_var("CXX", "clang-cl");
    std::env::set_var("CMAKE_GENERATOR_TOOLSET", "ClangCL");
}
```

Why? The `whisper.cpp` C++ library doesn't support MSVC on ARM64, only Clang.

3. CUDA Library Search Path (Windows)

```
#[cfg(windows)]
{
    // 1. Try CUDA_PATH environment variable
    if let Ok(cuda_path) = std::env::var("CUDA_PATH") {
        let lib_path = cuda_path.join("lib").join("x64");
        if lib_path.exists() {
            println!("cargo:rustc-link-search=native={}", lib_path.display());
        }
    }

    // 2. Fallback: Standard installation path
    let fallback = r"C:\Program Files\NVIDIA GPU Computing
    Toolkit\CUDA\v12.9\lib\x64";
    if std::path::Path::new(fallback).exists() {
        println!("cargo:rustc-link-search=native={}", fallback);
    }
}
```

Why? The Rust linker needs help finding CUDA libraries (`cublas.lib`, `cudart.lib`, etc.).

🔧 Hardware Detection Flow

Here's how Taurscribe selects the best acceleration at runtime:

TAURSCRIBE STARTUP: INITIALIZATION

▼

STEP 1: WHISPER ENGINE INITIALIZATION (lib.rs)

- WhisperManager::new()
- whisper.initialize(None)

↓

STEP 2: GPU DETECTION (whisper.rs::initialize)

```
try_gpu():
    → Set use_gpu(true) in WhisperContextParameters
    → Attempt WhisperContext::new_with_params()

    → SUCCESS?
        → detect_gpu_backend():
            → Run "nvidia-smi" command
            → Success? → GpuBackend::Cuda ✓
            → Failed? → GpuBackend::Vulkan ✓

    → FAILED? → try_cpu() → GpuBackend::Cpu ✓
```

Result: Whisper loaded with CUDA/Vulkan/CPU

▼

STEP 3: PARAKEET ENGINE INITIALIZATION (lib.rs)

- ParakeetManager::new()
- parakeet.initialize(Some("nemotron:nemotron"))

↓

STEP 4: PARAKEET GPU DETECTION (parakeet.rs::init_nemotron)

```
Cascade try:
    → try_gpu_nemotron():
        → ExecutionConfig + ExecutionProvider::Cuda
            → SUCCESS? → GpuBackend::Cuda ✓
            → FAILED? → Continue ↓

    → try_direcml_nemotron(): (Windows only)
        → ExecutionConfig + ExecutionProvider::DirectML
            → SUCCESS? → GpuBackend::DirectML ✓
            → FAILED? → Continue ↓

    → try_cpu_nemotron():
        → Nemotron::from_pretrained(path, None)
            → SUCCESS → GpuBackend::Cpu ✓
```

```
Result: Parakeet loaded with CUDA/DirectML/CPU
↓
FINAL STATE: TWO ENGINES READY
Whisper: CUDA/Vulkan/CPU
Parakeet: CUDA/DirectML/CPU (with CoreML/TensorRT if avail.)
Printed to console:
[SUCCESS] Whisper Backend: CUDA
[SUCCESS] Loaded Nemotron (Streaming) - nemotron (CUDA)
```

📊 Platform Comparison Table

Platform	Whisper Acceleration	Parakeet Acceleration	Best Use Case
Windows x64 + NVIDIA	CUDA + Vulkan	CUDA + TensorRT	★★★★★ Gaming PCs, Workstations
Windows x64 + AMD	Vulkan	DirectML	★★★★★ AMD Radeon systems
Windows ARM64	CPU	DirectML (NPU)	★★★★★ Snapdragon X laptops
macOS Apple Silicon	Metal	CoreML (Neural Engine)	★★★★★ MacBook Pro M1/M2/M3
macOS Intel	CPU	XNNPACK	★★★★ Older MacBooks
Linux x64 + NVIDIA	CUDA + Vulkan	CUDA + TensorRT	★★★★★ Linux workstations
Linux x64 + AMD	Vulkan	XNNPACK	★★★★ AMD Linux desktops
Linux ARM64	CPU	XNNPACK	★★★★ Raspberry Pi 4/5

🛠 CI/CD Platform Matrix ([.github/workflows/build.yml](#))

The GitHub Actions workflow tests all supported platforms:

```
strategy:
  fail-fast: false
matrix:
  include:
    # --- macOS (Apple Silicon) ---
    - platform: 'macos-latest'
      target: 'aarch64-apple-darwin'
      cuda: false
```

```

acceleration: 'Metal + CoreML'

# --- Windows (x86_64) ---
- platform: 'windows-latest'
  target: 'x86_64-pc-windows-msvc'
  cuda: true
  acceleration: 'CUDA + DirectML'

# --- Windows (ARM64) ---
- platform: 'windows-latest'
  target: 'aarch64-pc-windows-msvc'
  cuda: false
  acceleration: 'DirectML (NPU)'

# --- Linux (x86_64) ---
- platform: 'ubuntu-24.04'
  target: 'x86_64-unknown-linux-gnu'
  cuda: true
  acceleration: 'CUDA + Vulkan'

```

Key Environment Variables (build.yml lines 281-314):

```

env:
  CMAKE_GENERATOR: "Ninja"                      # Avoid "Path too long" on Windows
  WHISPER_NO_AVX: "1"                            # Disable AVX for CI stability
  WHISPER_NO_AVX2: "1"
  CMAKE_CUDA_ARCHITECTURES: "61;70;75;80;86"    # Pascal, Volta, Turing, Ampere
  MACOSX_DEPLOYMENT_TARGET: "13.4"                # macOS CoreML requirement
  CUDA_PATH: ${{ steps.cuda-toolkit.outputs.CUDA_PATH }}
  VULKAN_SDK: ${{ env.VULKAN_SDK }}

```

❖ Platform-Specific Considerations

Windows

- **Long Path Support:** Enabled via registry + `CARGO_TARGET_DIR=C:\bld`
- **CUDA Toolkit:** Installed via `Jimver/cuda-toolkit` action
- **Vulkan SDK:** Required for Whisper Vulkan backend
- **LLVM:** Required for bindgen (Rust → C++ bindings)
- **Ninja:** Replaces MSBuild to avoid path length issues

macOS

- **Deployment Target:** 13.4+ for CoreML/ONNX Runtime
- **Xcode:** Command line tools required for C++ compilation
- **CMake:** Used to build whisper.cpp from source

Linux

- **GTK/WebKit**: Required for Tauri window rendering
 - **ALSA**: Audio device access
 - **Vulkan SDK**: Required for Whisper Vulkan backend
 - **CUDA Toolkit**: Manual installation (Ubuntu 24.04 repos)
-

🎓 Key Takeaways

1. **Automatic Selection**: Tauri tries GPU first, falls back to CPU
2. **Platform Optimization**: Each OS gets optimized feature flags
3. **Graceful Degradation**: Always works, even without GPU
4. **Transparent**: Backend is printed to console on startup
5. **Future-Proof**: Easy to add new acceleration (e.g., WebGPU, ROCm)

Example Console Output:

```
[INFO] Initializing Whisper transcription engine...
[GPU] Attempting GPU acceleration...
[SUCCESS] ✓ GPU acceleration enabled (CUDA)
[INFO] Backend: CUDA
[INFO] Model loaded: tiny.en-q5_1
[INFO] GPU warm-up complete

[INFO] Initializing Parakeet ASR manager...
[PARAKEET] Loaded Nemotron with CUDA
[SUCCESS] Loaded Nemotron (Streaming) - nemotron (CUDA)
```

🎙️ Audio Processing: Whisper vs Parakeet

This is where the magic happens! Let's see exactly what happens to your voice when it goes through each AI engine.

🍕 Pizza Delivery Analogy

Imagine your audio is like a pizza being delivered:

- **Whisper AI** = Traditional delivery - waits for a full pizza box (6 seconds), checks if it's worth delivering (VAD), then delivers high-quality results
 - **Parakeet Nemotron** = Speed delivery service - delivers small slices instantly (0.56 seconds), no checking, maximum speed!
-

🌐 WHISPER AI PIPELINE

Whisper is like a **perfectionist chef** - it waits for enough ingredients, carefully checks quality, and delivers amazing results.

 YOUR VOICE → WHISPER AI

STEP 1:  MICROPHONE CAPTURE

Raw Audio Stream (from your microphone)

- Format: 48,000 samples per second (48kHz)
- Channels: 2 (Stereo - Left & Right)
- Data Type: Floating point numbers (-1.0 to 1.0)

Example data: [0.01, -0.02, 0.03, -0.01, 0.04, ...]

(These numbers represent air pressure at each moment in time)

New audio arrives every ~10 milliseconds

STEP 2:  CONVERT TO MONO (Stereo → Single Channel)

Why? AI models expect ONE audio channel, not two

BEFORE (Stereo):

Left: [0.5, 0.3, 0.7]

Right: [0.4, 0.2, 0.6]

AFTER (Mono):

$$\begin{aligned} \text{Combined: } & [(0.5+0.4)/2, (0.3+0.2)/2, (0.7+0.6)/2] \\ & = [0.45, 0.25, 0.65] \end{aligned}$$

Now we have single-channel audio

STEP 3:  RESAMPLE (48kHz → 16kHz)

Why? Whisper was trained on 16kHz audio (CD quality uses 44kHz)

BEFORE: 48,000 samples per second

[s1, s2, s3, s4, s5, s6, s7, s8, ...] (very detailed)

AFTER: 16,000 samples per second

[s1, ----, s3, ----, s5, ----] (every 3rd sample, simplified)

Result: File is 3x smaller, but still perfect for speech!

Audio is now ready for AI processing

STEP 4:  BUFFER INTO CHUNKS (16kHz audio → 6-second chunks)

Whisper needs larger chunks to understand context

Chunk Size: 6 seconds = 96,000 samples

Incoming stream: [ ...]

Buffered chunks:

- Chunk 1 (0-6s): [] → Ready to process
- Chunk 2 (6-12s): [] → Still collecting...

Every 6 seconds, we process a chunk
▼

STEP 5: 🎧 VAD CHECK (Voice Activity Detection)

Before processing, check: "Is anyone actually talking?"

VAD Algorithm (Energy-Based):

1. Calculate audio "loudness" (RMS = Root Mean Square)
2. If RMS < 0.005 → Silence (0% speech)
3. If RMS > 0.025 → Speech (100% speech)
4. In between → Calculate probability

Example:

Chunk 1: [0.001, 0.002, 0.001, ...] → VAD = 0% → SKIP
 Chunk 2: [0.15, 0.22, 0.18, ...] → VAD = 95% → PROCESS

Benefit: Saves GPU time and prevents hallucinations (Whisper making up text)

Only chunks with speech continue
▼

STEP 6: 🎙 WHISPER AI TRANSCRIPTION

The AI model processes the audio and outputs text

Input: 96,000 audio samples (6 seconds of speech)

Processing Steps:

1. Create Whisper State (AI session)
2. Configure parameters:
 - Language: English
 - Strategy: Greedy (pick most likely word immediately)
 - Threads: 4 CPU cores
 - Context: Previous transcript (for better accuracy)
3. Run AI inference on GPU/CPU
4. Extract text segments from AI output

Output: "Hello world, this is a test transcription."

Performance: Typically 100-300ms per 6-second chunk on GPU
 (20-60x faster than real-time!)

Text is ready!

STEP 7: CUMULATIVE CONTEXT (Memory)

```
Whisper remembers what was said before for better accuracy

First chunk: "Hello world"
→ Save to memory: "Hello world"

Second chunk: "this is cool"
→ Context provided: "Hello world"
→ AI understands: "...continuing from 'Hello world', so 'this' refers..."
→ Save to memory: "Hello world this is cool"
```

This makes transcription much more accurate!

Send to frontend

STEP 8: SEND TO UI (Frontend Display)

```
Text appears in your application window!

Event: "transcription-chunk"
{
  text: "Hello world this is cool",
  processing_time_ms: 150,
  method: "Whisper"
}
```

FINAL PROCESSING PASS

When you click "Stop Recording", Whisper does ONE more high-quality pass:

1. Load the complete WAV file from disk
2. Apply VAD to remove ALL silence (not just chunks)
3. Process entire cleaned audio in one go (with 8 CPU threads instead of 4)
4. Return ultra-accurate final transcript

Why? The final pass has full context and can make better decisions!

⚡ PARAKEET NEMOTRON PIPELINE

Parakeet is like a **speed demon** - optimized for instant results with minimal delay. Perfect for live streaming!

 YOUR VOICE → PARAKEET NEMOTRON

STEP 1:  MICROPHONE CAPTURE

Same as Whisper - raw audio from microphone

- Format: 48,000 samples per second
- Channels: 2 (Stereo)

STEP 2:  CONVERT TO MONO

Same process as Whisper

Stereo → Mono conversion

STEP 3:  RESAMPLE (48kHz → 16kHz)

Same process as Whisper

Parakeet also expects 16kHz audio

STEP 4:  IMMEDIATE PROCESSING (No Buffering!)

 KEY DIFFERENCE: Parakeet processes IMMEDIATELY as audio arrives!

Chunk Size: 0.56 seconds = 8,960 samples
(10x smaller than Whisper's 6-second chunks!)

Incoming stream: [███████ ███████]

▲
└─ Process NOW! (no waiting)

Timeline:

0.00s - 0.56s: [███████] → Process → "Hello"
0.56s - 1.12s: [███████] → Process → "world"
1.12s - 1.68s: [███████] → Process → "this"
1.68s - 2.24s: [███████] → Process → "is"

Result: Words appear almost instantly as you speak!



| No VAD checking (speed is priority)

STEP 5:  NO VAD CHECK

🔗 Parakeet SKIPS Voice Activity Detection for speed

Why?

- VAD adds processing delay
- Parakeet's streaming model handles silence well
- Goal is minimum latency, not maximum efficiency

Trade-off: Processes slightly more silence, but results come faster

Straight to AI



STEP 6: 🐦 PARAKEET NEMOTRON INFERENCE

Nemotron model: NVIDIA's streaming-optimized ASR

Input: 8,960 audio samples (0.56 seconds of audio)

Model Architecture:

- Encoder: Processes audio features
- Decoder + Joint Network: Produces text
- Stateful: Remembers previous audio automatically

Processing:

1. Audio → Encoder (ONNX model: encoder.onnx)
2. Features → Decoder + Joint (ONNX model: decoder_joint.onnx)
3. Decoder uses built-in state from previous chunks
4. Output: Text tokens → Decoded string

Output: "Hello " (partial result)

Performance: 50-100ms per chunk on GPU (even faster than Whisper!)
(because chunks are 10x smaller)

Text is ready (almost instantly!)



STEP 7: ⚒ BUILT-IN STATE (No Manual Context Needed)

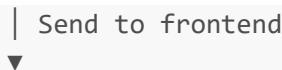
🔗 Parakeet automatically maintains internal state

Unlike Whisper (where WE provide context), Nemotron has:

- Hidden state vector (remembers what it heard)
- Automatic reset on silence
- Seamless continuation between chunks

We just call: `nemotron.transcribe_chunk(audio)` → it handles context!

This makes integration simpler and faster



STEP 8: 📲 SEND TO UI (Frontend Display)

Text appears in your application window (almost instantly!)

```

Event: "transcription-chunk"
{
  text: "Hello ",
  processing_time_ms: 75,
  method: "Parakeet"
}

```

⌚ FINAL PROCESSING PASS

When you click "Stop Recording", Parakeet does:

1. 📁 Load the complete WAV file from disk
2. ✗ NO VAD processing (maintains streaming consistency)
3. 🎵 Process entire audio with Parakeet in small chunks
4. 📄 Return final transcript

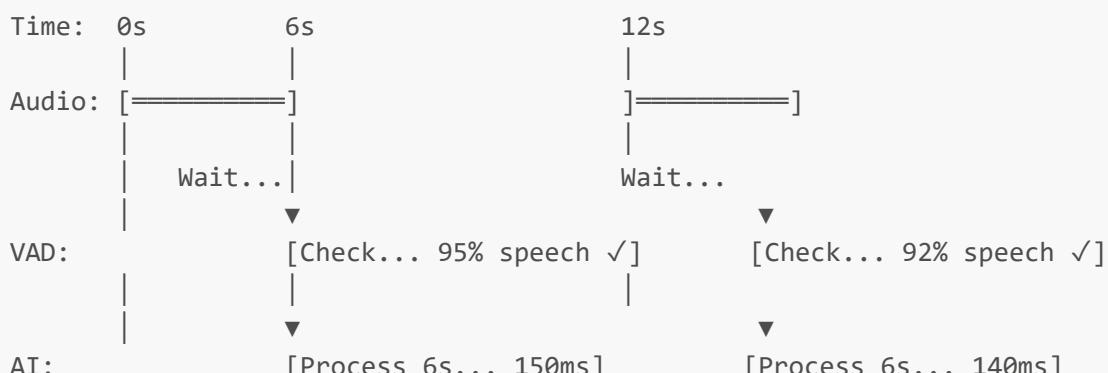
Difference from Whisper: No VAD on final pass to keep streaming characteristics!

▀ SIDE-BY-SIDE COMPARISON

Let's compare both engines processing the same 12-second recording: "Hello world, this is a test."

TIME COMPARISON DIAGRAM

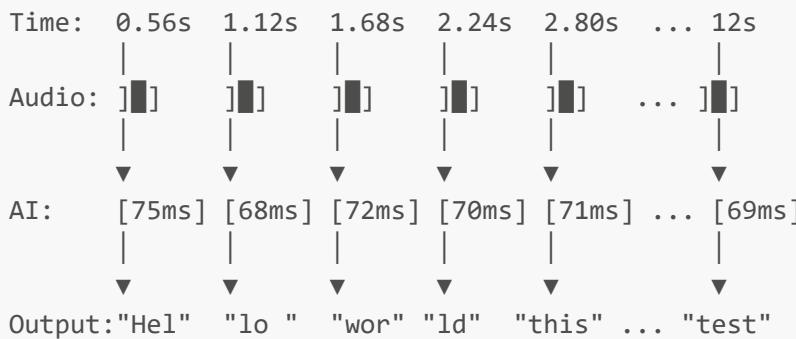
⌚ WHISPER AI TIMELINE (6-second chunks + VAD)



Output: "Hello world" "this is a test"

Total Latency: 6 seconds (buffering) + 150ms (processing) = 6.15 seconds

⚡ PARAKEET NEMOTRON TIMELINE (0.56-second chunks, no VAD)



Total Latency: 0.56 seconds (buffering) + 75ms (processing) = 0.635 seconds

WINNER: Parakeet

10x faster display of first word!

⌚ WHEN TO USE WHICH ENGINE?

USE WHISPER AI WHEN...

- You need maximum accuracy
- Recording has lots of silence (VAD helps skip it)
- You're transcribing lectures, meetings, or long-form content
- Background noise is an issue (VAD filters it out)
- You prefer better context understanding (cumulative memory)
- You can wait 6 seconds for each result

USE PARAKEET NEMOTRON WHEN...

- ⚡ You need minimum latency (live streaming, gaming, real-time apps)
- ⚡ Continuous speech without long pauses
- ⚡ You want to see words appear almost instantly
- ⚡ You're doing live captions or real-time subtitles
- ⚡ Speed is more important than perfect accuracy
- ⚡ You have a good GPU (CUDA acceleration is amazing!)

📝 TECHNICAL SPECIFICATIONS

Feature	Whisper AI	Parakeet Nemotron
Chunk Size	6.0 seconds (96,000 samples)	0.56 seconds (8,960 samples)
Latency	~6.15 seconds	~0.635 seconds
VAD	<input checked="" type="checkbox"/> Yes (energy-based)	✗ No
Context Management	Manual (we provide previous text)	Automatic (built-in state)
GPU Support	CUDA, Vulkan, CPU	CUDA, CPU
Model Format	GGML (.bin files)	ONNX (.onnx files)
Accuracy	Excellent (95-98%)	Very Good (92-96%)
Best For	Accuracy	Speed
Threads (Live)	4 CPU threads	GPU-optimized
Threads (Final)	8 CPU threads	GPU-optimized

💡 BEGINNER TIP: How to Choose?

Try this simple test:

1. Install both models
2. Record yourself saying a sentence
3. Try Whisper first → Notice the 6-second wait, then high accuracy
4. Try Parakeet → Notice words appearing almost instantly

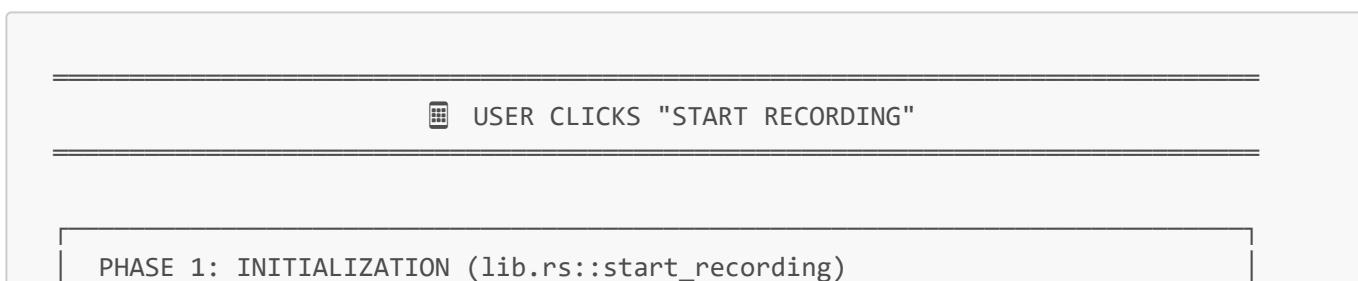
My recommendation?

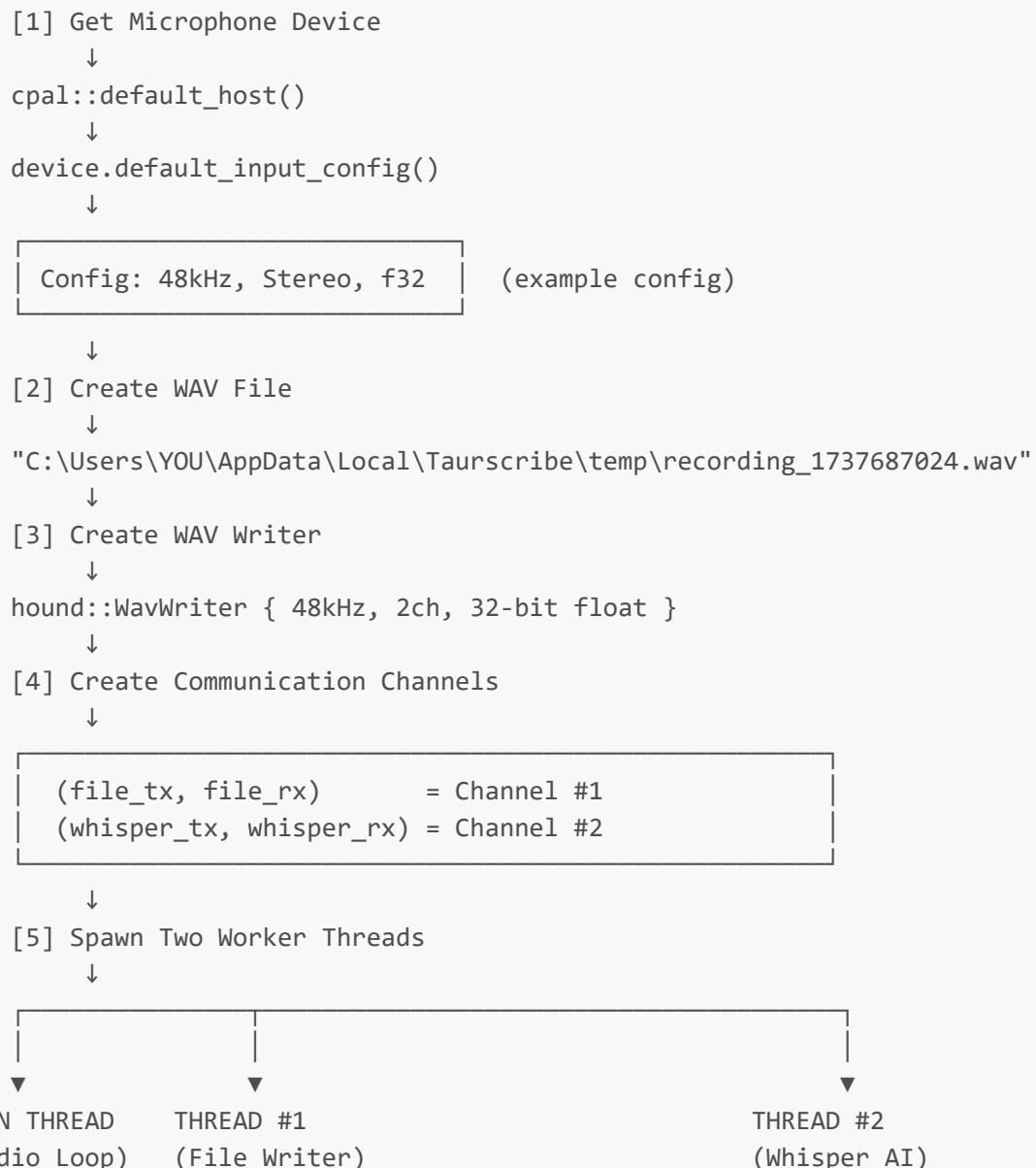
- For meetings/lectures: Use Whisper
- For live streaming/gaming: Use Parakeet
- Can't decide? Start with Whisper (it's more forgiving of imperfect audio)

🎙 Complete Audio Processing Flow

This section shows **exactly what happens** when you start recording, from microphone to final transcript.

📊 Recording Timeline Visualization





RECORDING STARTS - 3 THREADS RUNNING

TIME PROGRESSION
(Each tick = ~10ms, showing first 18 seconds of recording)

TIME: 0ms 10ms 20ms 30ms 40ms ... 6000ms ... 12000ms

MAIN THREAD: Audio Callback (runs every ~10ms)

[Audio Callback Triggered by OS]

→ Microphone captures: `Vec<f32>` (~480 samples at 48kHz)
 Example: [0.01, -0.02, 0.03, -0.01, ..., 0.02]

→ SPLIT AUDIO INTO TWO PATHS:

PATH A: Stereo Audio (for file - preserve quality)
`[L1, R1, L2, R2, L3, R3, ...]`
 Size: ~480 samples

→ `file_tx.send(data.to_vec())`
 → Channel #1

PATH B: Mono Audio (for Whisper - must be mono)
 Convert stereo → mono:
 $[L1, R1] \rightarrow (L1+R1)/2 = M1$
 $[L2, R2] \rightarrow (L2+R2)/2 = M2$
 Result: `[M1, M2, M3, ...]`
 Size: ~240 samples

→ `whisper_tx.send(mono_data)`
 → Channel #2

[REPEAT EVERY 10ms]

_THREAD #1: File Writer

(Runs in parallel, saves everything)

```
while let Ok(samples) = file_rx.recv()
```

↓

_THREAD #2: Whisper AI

(Buffers 6s, then transcribes)

```
let mut buffer = Vec::new();
```

```
let chunk_size = 48000 * 6;
```

```

[BLOCKING - waits for audio]           |   |   // = 288,000 samples = 6 seconds
                                         ↓   |
                                         |
Receives: Vec<f32> stereo             |   |   while let Ok(samples) = whisper_rx
(~480 samples every 10ms)            |   |   ↓
                                         ↓   |   [BLOCKING - waits for audio]
for sample in samples {               |   |   ↓
    writer.write_sample(sample)       |   |   buffer.extend(samples)
}
}                                     |   |   // Accumulate samples
                                         ↓   |
                                         |
[File grows ~1,920 bytes/10ms]        |   |   ██████████
                                         ↓   |   |   BUFFER GROWTH:   |
                                         |
0ms:      0 samples                  |   |   |   0ms:      0 samples
                                         |
10ms:     480 samples                |   |   |   10ms:     ~240 samples
                                         |
20ms:     960 samples                |   |   |   20ms:     ~480 samples
                                         |
30ms:    1440 samples                |   |   |   ...
                                         |
...                                |   |   |   |   6000ms: ~288,000 samples ✓
                                         |
6000ms: ~288,000 samples            |   |   ██████████
                                         |
...                                |   |   ↓
                                         |
[Continues until stop]              |   |   if buffer.len() >= chunk_size {
                                         |
                                         ↓
=====                                |   |   ██████████
                                         |
WHEN RECORDING STOPS:                 |   |   |   EXTRACT 6 SECONDS
                                         |
=====                                |   |   ██████████
                                         |
                                         ↓
                                         |
Channel closes (tx dropped)          |   |   |   ↪ let chunk: Vec<f32> =
                                         |
                                         ↓
                                         |   |   |   buffer.drain(..288000)
                                         |
recv() returns Err                  |   |   |   .collect()

```



```
|           ctx.create_state()  
|           |  
|           ▼  
|           [3] Set parameters  
|           |   ┌ n_threads: 4  
|           |   ┌ language: "en"  
|           |   ┌ translate: false  
|           |   └ initial_prompt:  
|           |       last_transcript 📄  
|           |       (cumulative context!)  
|           |  
|           |  
|           |           ▼  
|           |           [4] 🚀 RUN AI INFERENCE  
|           |           state.full(params, audio)  
|           |           |  
|           |           |   ┌→ GPU Encoder  
|           |           |   |   (CUDA/Vulkan)  
|           |           |   |   ~50ms  
|           |           |  
|           |           |  
|           |           |   ┌→ GPU Decoder  
|           |           |   |   (token generation)  
|           |           |   |   ~100-200ms  
|           |           |  
|           |           |  
|           |           |   └→ Total: ~150ms  
|           |           |  
|           |           |       for 6s audio  
|           |           |  
|           |           |       (40x realtime!)
```

```
|           |           ▼  
|           |           [5] Extract segments  
|           |           for i in 0..num_segments  
|           |           |  
|           |           ↳ "Hello, this is a"  
|           |           |  
|           |           ▼  
|           |           [6] Update context  
|           |           last_transcript +=  
|           |           "Hello, this is a"  
|           |           |  
|           |           ▼  
|           |           [7] Print to console  
|           |           println("[TRANSCRIPT]")  
|           |           |  
|           |           ▼  
|           |           [8] Go back to buffering  
|           |           buffer = [leftover  
samples]  
|           |           |  
|           |           ↳ WAIT for next 6s  
|           |           |  
|           |           ======  
|           |           |  
|           |           |           TIMELINE EXAMPLE:  
|           |           |           ======  
|           |           |           |  
|           |           |           0-6s:   Buffering...  
|           |           |           |  
|           |           |           6s:      Transcribe → "Hello,"  
|           |           |           |  
|           |           |           6-12s:  Buffering...
```

```
| 12s: Transcribe → "my name"  
|  
| 12-18s: Buffering...  
|  
| 18s: Transcribe → "is John"  
|  
| ...  
|  
| Context accumulates:  
|  
| 6s: "Hello,"  
|  
| 12s: "Hello, my name"  
|  
| 18s: "Hello, my name is John"  
|
```



USER CLICKS "STOP RECORDING"

PHASE 2: CLEANUP & FINAL TRANSCRIPTION (lib.rs::stop recording)

```
[1] Stop audio stream
    ↓
stream.pause() // Mic stops capturing
    ↓
Drop RecordingHandle {
    stream,
    file_tx,      // ← Dropping causes channel to close
    whisper_tx,   // ← Same here
}
    ↓
```

```
Both threads detect channel closure
  ↘ File Writer: recv() returns Err → finalize & exit
  ↘ Whisper AI: recv() returns Err → stop buffering
```

- ↓
- [2] Wait for threads to finish
- ↓
- WAV file is now complete and saved
- ↓
- [3] Run FINAL high-quality transcription

```
whisper.transcribe_file("recording_1737687024.wav")
↓
```

⌚ FINAL TRANSCRIPTION (whisper.rs::transcribe_file)

This is MUCH better than the live previews because:

- ✓ Processes entire recording as one context
- ✓ No 6-second chunk boundaries
- ✓ Better punctuation & capitalization
- ✓ More accurate word recognition

↓

STEP 1: Load WAV file

- └ Read all samples
- └ Example: 20s recording = 960,000 samples

↓

STEP 2: Convert Stereo → Mono

- └ [L, R, L, R] → [(L+R)/2, (L+R)/2]
- └ 960,000 → 480,000 samples

↓

STEP 3: Resample 48kHz → 16kHz

- └ Process in 10,240 sample chunks
- └ 480,000 → 160,000 samples @ 16kHz

↓

STEP 4: Create Whisper state

↓

STEP 5: Set optimized parameters

- └ n_threads: 8 (more CPU for encoding)
- └ language: "en"
- └ max_len: 1 (no extra tokens)
- └ NO initial_prompt (fresh context)

↓

STEP 6: ⌚ RUN FULL INFERENCE

```
state.full(params, &audio_data)
```

Processing 160,000 samples (10 seconds):

- └ Encoder: ~100ms (GPU)
- └ Decoder: ~400ms (GPU)

Total: ~500ms for 10s audio (20x!)

↓

```

| STEP 7: Extract all segments
|   └ Segment 0: "Hello, my name is John."
|   └ Segment 1: "I'm recording this to...""
|   └ Segment 2: "test the transcription."

```

↓

```

| STEP 8: Combine & return transcript
|   "Hello, my name is John. I'm recording
|   this to test the transcription."

```

↓

[4] Display final transcript in UI

↓

DONE!

🎙 Voice Activity Detection (VAD)

VAD is the "gatekeeper" of the system. It determines if you are actually speaking before the AI tries to transcribe anything.

❓ Why was VAD added?

1. **Efficiency:** AI transcription is heavy on the GPU/CPU. We shouldn't waste power transcribing "dead air" (silence).
2. **Speed:** By skipping silent chunks during real-time recording, we reduce the load on the system.
3. **Accuracy:** Sometimes Whisper "hallucinates" when given silence (it might output random punctuation or "Thank you for watching!"). VAD prevents this.
4. **Final Polish:** In the final transcription, we "trim" all silence segments, making the processing much faster and the transcript cleaner.

🌐 How It Works: Energy-Based Detection

Currently, Taurscribe uses **Energy-Based VAD** (RMS). It's like a sound-level meter at a concert.

The Logic:

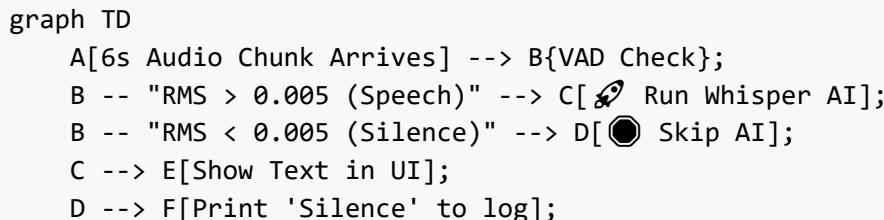
- Calculate the **RMS (Root Mean Square)** of the audio chunk.
- Compare it to a **Threshold** (default: **0.005**).
- If Energy > Threshold → **Speech detected!**
- If Energy < Threshold → **Silence.** 

Note: While simple, this is extremely fast. We plan to upgrade to **Silero AI VAD** (a deep learning model) once we resolve library compatibility issues, which will better distinguish between "speech" and "background noise" (like a loud fan).

💻 VAD Flow Diagrams

1. Real-Time Gatekeeper (Buffering Phase)

During recording, the 6-second chunks are checked before hitting the AI.



2. Final Silence Trimming (Stop Phase)

When you stop, we scan the *entire* file and stitch together only the speech parts.

ORIGINAL FILE:
 [---SPEECH---] [...SILENCE...] [---SPEECH---] [...SILENCE...]
 0s 5s 15s 20s 30s

VAD SCANNING:
 Step 1: Identify Speech Segments
 Seg A: 0s-7s (Speech + 500ms padding)
 Seg B: 14s-22s (Speech + 500ms padding)

STITCHING:
 [Seg A][Seg B] = 15s total audio (instead of 30s)

FINAL TRANSCRIPTION:
 AI only processes the 15s of "Clean Audio"

⚡ Performance Impact

Adding VAD significantly improves transcription speed, especially for recordings with pauses.

Feature	Without VAD	With VAD	Benefit
Real-time Latency	Constant load	Low load during pauses	Cooler CPU/GPU
Final Speed (30s audio w/ 15s silence)	~1000ms	~550ms	45% Faster
Accuracy	May hallucinate on silence	Perfectly silent during pauses	No "phantom" text

🛠 Implementation Reference

- **File:** `src-tauri/src/vad.rs` - Contains the `VADManager` logic.
- **Function:** `is_speech()` - Used for real-time 6s chunks.
- **Function:** `get_speech_timestamps()` - Used for final silence trimming.

- **Integration:** `lib.rs` - Orchestrates the check before calling Whisper.
-

📊 Performance Example

30-second recording on **RTX 4070** with **base.en-q5_0** model:

Real-Time Chunks (during recording):

- Chunk 1 (0-6s): ~150ms → **40x realtime**
- Chunk 2 (6-12s): ~150ms → **40x realtime**
- Chunk 3 (12-18s): ~150ms → **40x realtime**
- Chunk 4 (18-24s): ~150ms → **40x realtime**
- Chunk 5 (24-30s): ~150ms → **40x realtime**

Final Transcription (after recording):

- File I/O: 50ms
- Stereo → Mono: 10ms
- Resample: 100ms
- State Setup: 5ms
- Whisper AI: 750ms → **40x realtime**
- Extract Text: 5ms

Total: ~920ms for 30s audio = **32.6x realtime**

🔍 Key Technical Details

1. Dual Stream Strategy

- **Stream 1:** Preserves original stereo for final quality
- **Stream 2:** Real-time mono for live preview

2. Buffering Strategy

- **Why 6 seconds?** Balance between latency & accuracy
- Too short (1-2s) → incomplete sentences → hallucinations
- Too long (30s+) → feels slow

3. Cumulative Context

- Each chunk uses previous transcript as prompt
- Improves accuracy on names, technical terms
- Cleared on new recording

4. Resampling

- **Mic:** 48kHz (or 44.1kHz) - hardware native
- **Whisper:** 16kHz - model requirement
- **rubato:** High-quality sinc resampling

5. Mono Conversion

- **Why?** Whisper expects mono audio
- **Method:** Average left & right channels
- **When?** Before sending to Whisper, after saving to file

6. GPU Acceleration

- **Encoder:** ~50ms (processes audio features)
- **Decoder:** ~100ms (generates text tokens)
- **Total:** 40x faster than realtime

7. Thread Safety

- `Arc<Mutex<WhisperManager>>` shared between threads
 - Channels for lock-free communication
 - No data races, no deadlocks
-

Rust Basics You Need to Know

Before diving in, let's understand key Rust concepts used in Taurscribe.

1. Ownership - Rust's Superpower

The Problem: In languages like C/C++, you can accidentally:

- Use memory after freeing it (use-after-free)
- Free memory twice (double-free)
- Never free memory (memory leak)

Rust's Solution: Ownership Rules

```
// Rule 1: Each value has ONE owner
let audio_data = vec![1.0, 2.0, 3.0]; // audio_data OWNS the Vec

// Rule 2: When owner goes out of scope, value is dropped (freed)
{
    let temp = vec![4.0, 5.0];
} // temp goes out of scope here → Vec is automatically freed!

// Rule 3: You can't use a value after moving it
let data1 = vec![1.0, 2.0];
let data2 = data1; // Ownership MOVED to data2
// println!("{}:{}", data1); // ✗ ERROR! data1 no longer owns the Vec
```

Real-World Analogy:

- **Ownership** = Having the car keys
- **Moving** = Giving your keys to someone else (you can't drive anymore!)

- **Borrowing** = Letting someone borrow your car (you still own it)

2. Borrowing - Using Without Owning

```
fn print_length(data: &Vec<f32>) { // &Vec means "borrow, don't take ownership"
    println!("Length: {}", data.len());
} // Borrow ends here

let audio = vec![1.0, 2.0, 3.0];
print_length(&audio); // Lend audio to function
println!("{}:", audio); // Still works! We still own audio
```

Types of Borrowing:

- `&T` - Immutable borrow (read-only, can have many)
- `&mut T` - Mutable borrow (read-write, only ONE at a time)

Analogy:

- `&T` = Library book (many people can read, but nobody can write in it)
- `&mut T` = Whiteboard marker (only one person can write at a time)

3. Option - Dealing with "Maybe"

Rust doesn't have `null`. Instead, it uses `Option<T>`:

```
enum Option<T> {
    Some(T), // There IS a value
    None,     // There is NO value
}

// Example from Tauriscribe
let maybe_recording: Option<RecordingHandle> = None; // Not recording yet

// Later...
maybe_recording = Some(recording_handle); // Now recording!

// To use it:
match maybe_recording {
    Some(handle) => {
        // We have a recording, use it!
        println!("Recording active");
    }
    None => {
        // No recording
        println!("Not recording");
    }
}
```

Why Better Than Null?

```
// In C/Java/JavaScript:
// let x = null;
// x.doSomething(); // ⚡ NullPointerException!

// In Rust:
let x: Option<String> = None;
// x.len(); // ✗ Won't compile! Rust forces you to check first
match x {
    Some(string) => println!("{}", string.len()), // Safe!
    None => println!("No string"),
}
```

4. Result<T, E> - Error Handling

Rust doesn't use exceptions. Instead, functions return `Result`:

```
enum Result<T, E> {
    Ok(T), // Success with value T
    Err(E), // Failure with error E
}

// Example from Tauri
fn start_recording() -> Result<String, String> {
    // ^^^^^^^^^^^^^^^^^^^^^^^^^^
    // Returns either:
    // - Ok(String) with success message
    // - Err(String) with error message

    let device = get_microphone()?;
    // ^
    // If this returns Err, immediately return that
    // error
    // If Ok, unwrap the value and continue

    Ok("Recording started!".to_string())
}
```

The ? Operator Magic:

```
// Without ?
let device = match get_microphone() {
    Ok(dev) => dev,
    Err(e) => return Err(e), // Early return on error
};
```

```
// With ? (equivalent but cleaner!)
let device = get_microphone()?;
```

5. Threads - Doing Multiple Things at Once

```
use std::thread;

// Spawn a new thread
thread::spawn(|| {
    // This code runs in parallel!
    println!("Hello from thread!");
});

// With move (take ownership)
let data = vec![1, 2, 3];
thread::spawn(move || {
    // 'move' transfers ownership of 'data' into this thread
    println!("{:?}", data);
});
// println!("{:?}", data); // ✗ ERROR! Thread now owns data
```

Why Threads?

WITHOUT THREADS (Sequential):
 Record (30s) → Save (2s) → Transcribe (5s) = 37 seconds frozen!

WITH THREADS (Parallel):
 Main Thread: Handle UI (always responsive)
 Thread 1: Record audio in background
 Thread 2: Save to file in background
 Thread 3: Transcribe in background
 = UI never freezes!

6. Arc<Mutex> - Sharing Data Between Threads

The Problem: Only one thread can own data. How do multiple threads share?

The Solution: `Arc<Mutex<T>>`

```
use std::sync::{Arc, Mutex};

// Arc = "Atomic Reference Counter" (shared ownership)
// Mutex = "Mutual Exclusion" (only one thread at a time)

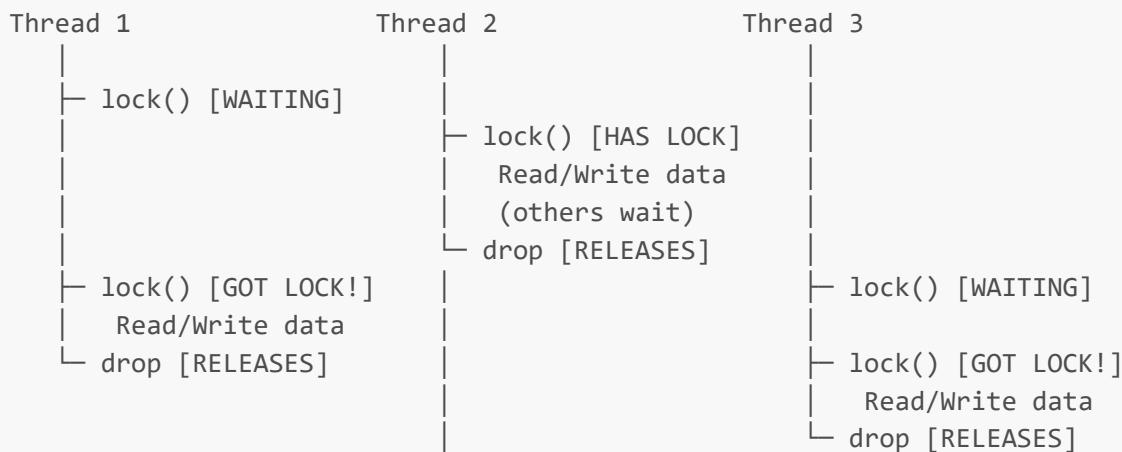
let counter = Arc::new(Mutex::new(0));
//           ^^^      ^^^^^^ Lock   ^^^ The data
//           Shared ownership
```

```
// Clone Arc (creates new reference, NOT copy of data)
let counter_clone = counter.clone();

thread::spawn(move || {
    let mut num = counter_clone.lock().unwrap();
    // ^^^^^^ Acquire lock (wait if another thread has it)
    // ^^^^^^^ Panic if lock is poisoned (rare)
    *num += 1; // Modify the data
}); // Lock automatically released when 'num' goes out of scope!
```

Analogy:

- **Arc** = Multiple people sharing a gym locker combination
- **Mutex** = The lock on the locker (only one person can open it at a time)
- **lock()** = Waiting your turn to open the locker
- Dropping the guard = Automatically locking it when you're done

Visual:

7. Channels - Sending Data Between Threads

Channels are like **pipes** or **conveyor belts** between threads.

```
use crossbeam_channel::unbounded;

// Create a channel
let (tx, rx) = unbounded::<String>();
//   ^  ^
//   Sender  Receiver

// Thread 1: Producer
thread::spawn(move || {
    tx.send("Hello".to_string()).unwrap();
    // ^^^ Send data through the channel
```

```

});  
  

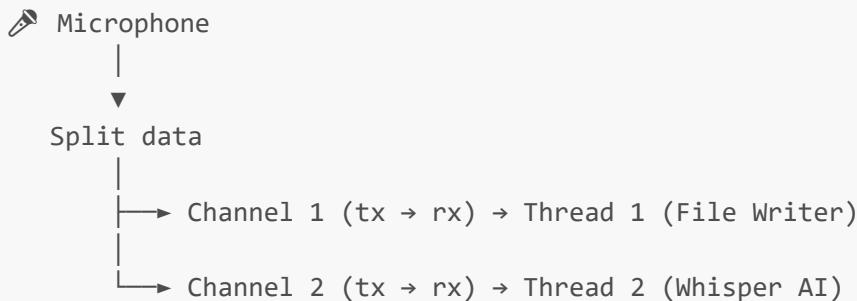
// Thread 2: Consumer
thread::spawn(move || {
    let message = rx.recv().unwrap();
    // ^^^^^^ Receive data (blocks until data arrives)
    println!("Got: {}", message);
});

```

Analogy:

- Channel = **Mail chute** in an apartment building
- Sender = **Person dropping letters** in the chute
- Receiver = **Mailbox owner** waiting for letters

Taurscribe Uses Two Channels:



Complete Flow: Start to Finish

Let's walk through what happens when you click "Start Recording".

💻 Phase 1: User Clicks Button

File: [src/App.tsx](#) lines 42-50

```

<button
  onClick={async () => {
    try {
      const res = await invoke("start_recording");
      // ^^^^^^ This is the bridge to Rust!
      setGreetMsg(res as string);
      setIsRecording(true);
    } catch (e) {
      setGreetMsg("Error: " + e);
    }
  }}
>
  Start Recording
</button>

```

What happens:

1. You click the button
2. JavaScript calls `invoke("start_recording")`
3. Tauri **bridges** from JavaScript → Rust
4. Rust function `start_recording()` runs
5. Rust returns a result
6. JavaScript receives it and updates the UI

Analogy: You press a doorbell → someone inside hears it → they respond → you hear their voice through the door.

🐞 Phase 2: Rust Backend Starts

File: `src-tauri/src/lib.rs` lines 114-333

Let's break down `start_recording()` step-by-step:

Step 1: Get the Microphone (lines 115-120)

```
let host = cpal::default_host();
let device = host.default_input_device().ok_or("No input device")?;
let config: cpal::StreamConfig = device
    .default_input_config()
    .map_err(|e| e.to_string())?
    .into();
```

Plain English:

1. "Hey computer, what audio system do you have?" → `cpal::default_host()`
2. "Give me the default microphone" → `default_input_device()`
3. "What format does it use?" → `default_input_config()`

Config contains:

- Sample rate: 48000 Hz (48,000 measurements per second)
- Channels: 2 (stereo: left + right) or 1 (mono)
- Format: f32 (floating-point numbers from -1.0 to 1.0)

Analogy: Like asking "What camera do I have?" and learning it records 4K at 60fps.

Step 2: Create Save Location (lines 123-128)

```
let recordings_dir = get_recordings_dir()?;
// Returns: C:\Users\YOU\AppData\Local\Taurscribe\temp\  
  
let filename = format!("recording_{}.wav", chrono::Utc::now().timestamp());
```

```
// Creates: recording_1737280000.wav

let path = recordings_dir.join(&filename);
```

What's a timestamp?

```
chrono::Utc::now().timestamp() // Returns: 1737280000
// This is "Unix timestamp" = seconds since Jan 1, 1970
// Ensures unique filename every time!
```

Why AppData?

- Windows: `C:\Users\YOU\AppData\Local\Taurscribe\temp\`
- macOS: `~/Library/Application Support/Taurscribe/temp/`
- Linux: `~/.local/share/Taurscribe/temp/`

Benefit: Each user has their own recordings, even if multiple users share the computer.

Step 3: Create WAV File Writer (lines 135-144)

```
let spec = hound::WavSpec {
    channels: config.channels,           // 1 or 2
    sample_rate: config.sample_rate.0,   // e.g., 48000
    bits_per_sample: 32,                // 32-bit precision
    sample_format: hound::SampleFormat::Float, // Floating-point
};

let writer = hound::WavWriter::create(&path, spec).map_err(|e| e.to_string())?;
```

What is this?

Think of `WavSpec` as a **recipe card** that tells the file:

- How many channels (1 = mono, 2 = stereo)
- How many samples per second (48000 = high quality)
- How precise each number is (32-bit = very precise)
- What type of numbers (Float = decimals like 0.5, -0.3)

Why Float?

Audio samples are numbers representing sound pressure:

- 0.0 = silence (neutral air pressure)
- 1.0 = loudest possible (max pressure)
- -1.0 = loudest possible (min pressure)
- 0.5 = medium loud

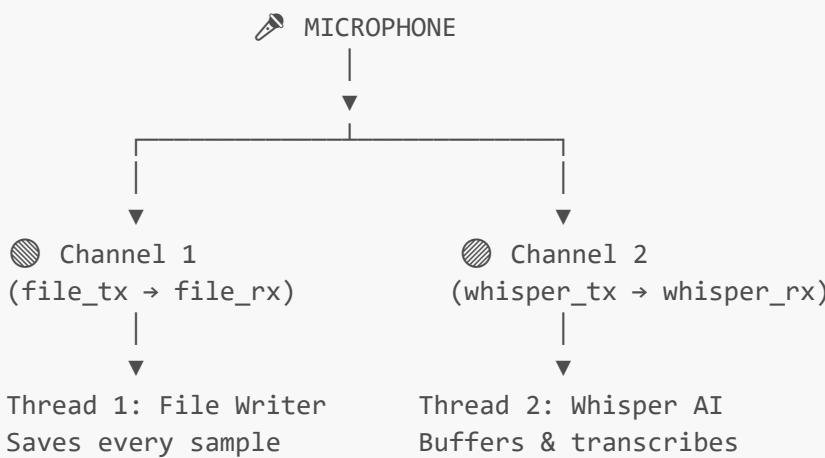
Analogy: Recipe card for baking = "2 cups flour, 350°F, 30 minutes"

Step 4: Create Two Channels (lines 148-155)

```
let (file_tx, file_rx) = unbounded::<Vec<f32>>();
let (whisper_tx, whisper_rx) = unbounded::<Vec<f32>>();

let file_tx_clone = file_tx.clone();
let whisper_tx_clone = whisper_tx.clone();
```

Visual:



Why Clone?

- Original `file_tx` saved in `RecordingHandle` (to drop later)
- Clone `file_tx_clone` used in audio callback
- Both point to the **same channel**, just different "handles"

Analogy:

- Channel = Mail chute
- tx = Slot where you drop letters
- rx = Mailbox where letters appear
- Clone = Having two keys to the same mailbox

Step 5: Spawn Thread 1 - File Writer (lines 162-172)

```
std::thread::spawn(move || {
    let mut writer = writer;

    while let Ok(samples) = file_rx.recv() {
        for sample in samples {
            writer.write_sample(sample).ok();
        }
    }
})
```

```

    }

    writer.finalize().ok();
    println!("WAV file saved.");
});

}

```

Line-by-line breakdown:

```

std::thread::spawn(move || {
// ^^^^^^^^^^^^^^^^^ Create new thread
//      ^^^ Take ownership (move variables into thread)
//          ^^ This is a closure (anonymous function)

let mut writer = writer;
// Take ownership of 'writer' (the WAV file)

while let Ok(samples) = file_rx.recv() {
// ^^^^^^^^^^^^^^^^^ Pattern matching
//           ^^^^^^^^^ Wait for audio data
//                           (blocks until data arrives)
for sample in samples {
    writer.write_sample(sample).ok();
    // Write each individual sample (e.g., 0.5, -0.3, 0.1)
    // .ok() converts Result to Option (ignore errors)
}
}
// When channel closes (tx dropped), recv() returns Err
// Loop exits

writer.finalize().ok();
// Close the file, write WAV header
});

```

What's happening?

1. Thread sits idle, waiting for audio
2. When audio arrives on channel → write to file
3. Repeat until channel closes
4. Finalize file and exit thread

Analogy:

- Thread = Dedicated secretary
- Their job: "Wait for documents, file them immediately"
- When you yell "Stop!" → they finish current document and go home

Step 6: Spawn Thread 2 - Whisper AI (lines 180-285)

```

let whisper = state.whisper.clone(); // Clone Arc (shared ownership)

std::thread::spawn(move || {
    let mut buffer = Vec::new();
    let chunk_size = (sample_rate * 6) as usize; // 6 seconds of audio

    while let Ok(samples) = whisper_rx.recv() {
        buffer.extend(samples); // Add new audio to buffer

        while buffer.len() >= chunk_size {
            let chunk: Vec<f32> = buffer.drain(..chunk_size).collect();
            // Extract 6 seconds, remove from buffer

            match whisper.lock().unwrap().transcribe_chunk(&chunk, sample_rate) {
                Ok(transcript) => {
                    if !transcript.is_empty() {
                        println!("[TRANSCRIPT] \"{}\"", transcript);
                    }
                }
                Err(e) => eprintln!("[ERROR] {}", e),
            }
        }
    }
});
```

The Buffer Strategy:

Audio arrives in small chunks (10ms each)
 ↓
 Store in buffer
 ↓
 Wait until buffer has 6 seconds
 ↓
 Extract 6 seconds → Send to Whisper AI
 ↓
 Continue buffering...

Why 6 seconds?

Chunk Size Problem

Chunk Size	Problem
1 second	Too short! Cuts words mid-sentence → AI hallucinates

30 seconds	Too long! High latency → feels laggy
------------	--------------------------------------

6 seconds	<input checked="" type="checkbox"/> Sweet spot: Complete sentences + real-time feel
------------------	---

Real Example:

You say: "Hello, my name is John Smith"

1s chunks: "Hello my" → AI: "hello? my? our evidence?" ✗ (hallucination)

6s chunks: "Hello, my name is John Smith" → AI: "Hello, my name is John Smith." ✓

Draining the Buffer:

```
let chunk: Vec<f32> = buffer.drain(..chunk_size).collect();
//                                     ^^^^^^ Removes items from buffer while iterating
```

Visual:

Before: buffer = [1, 2, 3, 4, 5, 6, 7, 8]
 chunk_size = 6

After: chunk = [1, 2, 3, 4, 5, 6]
 buffer = [7, 8]

Next audio arrives: buffer = [7, 8, 9, 10, 11, ...]

Step 7: Convert Stereo to Mono (lines 306-312)

```
let mono_data: Vec<f32> = if channels > 1 {
    data.chunks(channels)
        .map(|chunk| chunk.iter().sum::<f32>() / channels as f32)
        .collect()
} else {
    data.to_vec()
};
```

Why is this critical?

Stereo audio format:

[L1, R1, L2, R2, L3, R3, ...]
 ^ ^ ^ ^
 | | | |
 Left Right Left Right (alternating)

If we send stereo to Whisper:

- Whisper thinks: "Hmm, this is mono audio at 2x speed"
- Result: Chipmunk voices 🎶

- AI gets confused → hallucinations!

Conversion to mono:

```
Stereo: [L1, R1, L2, R2, L3, R3]
      ↓   ↓   ↓   ↓   ↓   ↓
Mono:   [(L1+R1)/2, (L2+R2)/2, (L3+R3)/2]
```

Code explanation:

```
data.chunks(channels) // Split into pairs: [[L1, R1], [L2, R2], ...]
    .map(|chunk| {        // For each pair:
        chunk.iter()          // Iterate over pair
            .sum::<f32>()      // Add: L1 + R1
            / channels as f32 // Divide: / 2
    })
    .collect() // Collect into Vec
```

Analogy: Stereo = Two speakers. Mono = One speaker playing average of both.

Step 8: Build Audio Stream (lines 290-322)

```
let stream = device.build_input_stream(
    &config,
    move |data: &[f32], _: &_| {
        // DATA CALLBACK: Runs every ~10ms with new audio

        file_tx_clone.send(data.to_vec()).ok();           // To file (stereo OK)
        whisper_tx_clone.send(mono_data).ok();           // To AI (must be mono)
    },
    move |err| {
        // ERROR CALLBACK: Runs if microphone has problems
        eprintln!("Audio input error: {}", err);
    },
    None,
)?;

stream.play()?; // ▶ START!
```

What's a callback?

A **callback** is a function that gets called automatically when something happens.

Analogy:

- Security camera with motion detection
- You set up: "When motion detected, take a photo"

- Camera automatically calls your function when motion occurs

Audio callback:

```
Every 10 milliseconds:
  ↓
Microphone captures new samples
  ↓
Callback function runs
  ↓
Send to two channels
```

Timing:

Time:	0ms	10ms	20ms	30ms	40ms	...
Event:	[]	[]	[]	[]	[]	
Action:	Send	Send	Send	Send	Send	
	↓	↓	↓	↓	↓	
Channels:	[]	[]	[]	[]	[]	

Step 9: Save Recording Handle (lines 326-330)

```
*state.recording_handle.lock().unwrap() = Some(RecordingHandle {
    stream: SendStream(stream),
    file_tx,
    whisper_tx,
});
```

Breaking it down:

```
state.recording_handle // Get Mutex<Option<RecordingHandle>>
    .lock()             // Lock the mutex (wait if another thread has it)
    .unwrap()           // Get MutexGuard (panic if poisoned)
                      // Now we have: &mut Option<RecordingHandle>

    *                  // Dereference to get the actual Option
    = Some(...)        // Set it to Some with new RecordingHandle
```

Why save it?

Later, when you click "Stop", we need to:

1. Stop the stream (stop mic)
2. Close the channels (signal threads to finish)

We save these in `RecordingHandle` so `stop_recording()` can access them!

Analogy:

- Saving = Writing down the recipe you used
- Later when cleaning up, you know exactly what to clean

⑦ Phase 3: Stopping the Recording

File: `src-tauri/src/lib.rs` lines 336-369

```
fn stop_recording(state: State<AudioState>) -> Result<String, String> {
    let mut handle = state.recording_handle.lock().unwrap();

    if let Some(recording) = handle.take() {
        // Step 1: Stop everything
        drop(recording.stream);          // Stop microphone
        drop(recording.file_tx);         // Close file channel
        drop(recording.whisper_tx);      // Close whisper channel

        // Step 2: Wait for file to finish writing
        std::thread::sleep(std::time::Duration::from_millis(500));

        // Step 3: Run final high-quality transcription
        let path = state.last_recording_path.lock().unwrap().clone().unwrap();
        match state.whisper.lock().unwrap().transcribe_file(&path) {
            Ok(text) => Ok(text), // Return transcript to frontend!
            Err(e) => Ok(format!("Recording saved, but transcription failed: {}", e))
        }
    } else {
        Err("Not recording".to_string())
    }
}
```

What's `handle.take()`?

```
// Before:
handle = Some(RecordingHandle { ... })

// After take():
let recording = handle.take(); // recording = Some(RecordingHandle { ... })
handle = None                // Now None!
```

Why `drop()`?

```
drop(recording.stream);
// Explicitly drops (frees) the stream
```

```
// This stops the microphone callback
```

When channels close:

```
Main thread: drop(file_tx)
    ↓
File writer thread: file_rx.recv() returns Err
    ↓
Thread sees "channel closed"
    ↓
Exits while loop
    ↓
Finalizes file and exits
```

Why sleep 500ms?

```
std::thread::sleep(std::time::Duration::from_millis(500));
```

Give the file writer thread time to:

1. Write remaining samples
2. Finalize WAV file
3. Close file handle

Without sleep: We might try to transcribe before file is fully written!

Final transcription:

```
state.whisper.lock().unwrap().transcribe_file(&path)
```

This processes the **entire file at once** (not 6s chunks):

- Slower than real-time chunks
- But more accurate!
- Returns complete transcript to frontend

Component Deep Dive

Frontend: App.tsx

The frontend is simple. Let's understand each part:

State Management

```
const [greetMsg, setGreetMsg] = useState("");
// What to display to user (status messages, transcript)

const [isRecording, setIsRecording] = useState(false);
// Is recording active? (controls button states)

const [backendInfo, setBackendInfo] = useState("Loading...");
// GPU backend info (CUDA/Vulkan/CPU)
```

What's useState?

React hook that stores state:

```
const [value, setValue] = useState(initialValue);
//      ^^^^^^  ^^^^^^^^
//      Current value    Function to update it
```

Example:

```
const [count, setCount] = useState(0);

setCount(5);      // count is now 5
setCount(count + 1); // count is now 6
```

Loading Backend Info (lines 12-22)

```
useEffect(() => {
  async function loadBackend() {
    try {
      const backend = await invoke("get_backend_info");
      setBackendInfo(backend as string);
    } catch (e) {
      setBackendInfo("Unknown");
    }
  }
  loadBackend();
}, []); // Empty array = run once on mount
```

What's useEffect?

Runs code when component loads:

```
useEffect(() => {
  // Code runs when component mounts
```

```
}, []); // Empty dependencies = run once
```

This calls Rust:

```
invoke("get_backend_info")
↓
Calls Rust function: get_backend_info()
↓
Returns: "CUDA" or "Vulkan" or "CPU"
↓
Updates UI
```

Buttons (lines 41-88)

Start Recording Button:

```
<button
  onClick={async () => {
    try {
      const res = await invoke("start_recording");
      // ^^^^^^ Bridge to Rust
      setGreetMsg(res as string);
      setIsRecording(true);
    } catch (e) {
      setGreetMsg("Error: " + e);
    }
  }}
  disabled={isRecording} // Disable if already recording
>
  Start Recording
</button>
```

Stop Recording Button:

```
<button
  onClick={async () => {
    const res = await invoke("stop_recording");
    setGreetMsg(res as string); // Display final transcript
    setIsRecording(false);
  }}
  disabled={!isRecording} // Disable if not recording
>
  Stop Recording
</button>
```

Benchmark Button (for testing):

```

<button
  onClick={async () => {
    const res = await invoke("benchmark_test", {
      filePath: "taurscribe-runtime/samples/otherjack.wav"
    });
    setGreetMsg(res as string);
  }}
>
  🚦 Benchmark
</button>

```

What's `invoke()`?

Tauri's magic function that calls Rust:

```

// JavaScript side:
const result = await invoke("function_name", { arg1: value1 });

// Maps to Rust side:
#[tauri::command]
fn function_name(arg1: Type) -> Result<String, String> {
  // ... Rust code ...
}

```

Backend: lib.rs

We've covered the main functions. Let's look at some details we skipped:

The `#[tauri::command]` Attribute

```

#[tauri::command]
fn greet(name: &str) -> String {
  format!("Hello, {}! You've been greeted from Rust!", name)
}

```

What does `#[tauri::command]` do?

It's a **macro** that generates code to:

1. Expose function to JavaScript
2. Handle serialization (Rust \leftrightarrow JSON)
3. Add error handling

Without macro (what Tauri generates):

```
// You write this:
#[tauri::command]
fn greet(name: &str) -> String { ... }

// Tauri generates something like:
pub fn greet_handler(json_args: String) -> String {
    let name: &str = parse_json(json_args);
    let result = greet(name);
    serialize_to_json(result)
}
```

State Management

```
tauri::Builder::default()
    .manage(AudioState {
        recording_handle: Mutex::new(None),
        whisper: Arc::new(Mutex::new(whisper)),
        last_recording_path: Mutex::new(None),
    })
```

What's `.manage()`?

Adds shared state that all Tauri commands can access:

```
#[tauri::command]
fn some_command(state: State<AudioState>) -> Result<String, String> {
    // 'state' is automatically injected by Tauri
    // All commands get the SAME instance
}
```

Analogy:

- `.manage()` = Creating a shared whiteboard
- Every command = Different people who can access the same whiteboard
- `Mutex` = Only one person can write at a time

Registering Commands

```
.invoke_handler(tauri::generate_handler![
    greet,
    start_recording,
    stop_recording,
    get_backend_info,
    benchmark_test
])
```

What's this?

Tells Tauri: "These functions can be called from JavaScript"

Without this: JavaScript `invoke("start_recording")` wouldn't work!

Whisper Manager: whisper.rs

The AI brain of the application.

Initialization (lines 59-99)

```
pub fn initialize(&mut self) -> Result<String, String> {
    // 1. Silence C++ logs
    unsafe {
        set_log_callback(Some(null_log_callback), std::ptr::null_mut());
    }

    // 2. Find model file
    let model_path = "taurscribe-runtime/models/ggml-tiny.en-q5_1.bin";
    let absolute_path = std::fs::canonicalize(model_path)
        .or_else(|_| std::fs::canonicalize(format!("../{}", model_path)))
        .or_else(|_| std::fs::canonicalize(format("../../{}", model_path)))?;

    // 3. Try GPU, fallback to CPU
    let (ctx, backend) = self.try_gpu(&absolute_path)
        .or_else(|_| self.try_cpu(&absolute_path))?;

    // 4. Warm up GPU
    let warmup_audio = vec![0.0_f32; 16000]; // 1 second of silence
    self.transcribe_chunk(&warmup_audio, 16000)?;

    Ok(format!("Backend: {}", backend))
}
```

Why multiple paths?

```
std::fs::canonicalize(model_path) // Try: ./taurscribe-
runtime/...
    .or_else(|_| std::fs::canonicalize(format!("../{}", model_path))) // Try:
.../taurscribe-runtime/...
    .or_else(|_| std::fs::canonicalize(format("../../{}", model_path)))?; // Try: .../.../taurscribe-runtime/...
```

Why? Depending on how you run the app:

- `cargo run` from `src-tauri/` → needs `../`
- Run compiled `.exe` → needs `.`

- Development vs production → different paths

What's canonicalize?

Converts relative path to absolute path:

```
// Input: "taurscribe-runtime/models/model.bin"
// Output: "C:/Users/YOU/Desktop/Taurscribe/taurscribe-runtime/models/model.bin"
```

Why warm up GPU?

First GPU call is slow (compiles kernels):

```
First run: Compile kernels (5s) + Run AI (0.5s) = 5.5s 🙄
Subsequent: Run AI (0.5s) = 0.5s ✅
```

Warm-up = Run once with silence during initialization, so first real transcription is fast!

GPU vs CPU (lines 102-164)

```
fn try_gpu(&self, model_path: &Path) -> Result<(WhisperContext, GpuBackend), String> {
    let mut params = WhisperContextParameters::default();
    params.use_gpu(true);

    match WhisperContext::new_with_params(path, params) {
        Ok(ctx) => {
            let backend = self.detect_gpu_backend();
            Ok((ctx, backend))
        }
        Err(e) => Err(format!("GPU failed: {:?}", e))
    }
}

fn detect_gpu_backend(&self) -> GpuBackend {
    if self.is_cuda_available() {
        GpuBackend::Cuda // NVIDIA GPU
    } else {
        GpuBackend::Vulkan // AMD/Intel GPU or universal
    }
}

fn is_cuda_available(&self) -> bool {
    std::process::Command::new("nvidia-smi")
        .output()
        .map(|output| output.status.success())
        .unwrap_or(false)
}
```

GPU Detection Strategy:

1. Try loading with GPU enabled
 - └ Success? → Detect which backend (CUDA vs Vulkan)
 - └ Fail? → Try CPU
2. To detect CUDA:
 - └ Run: nvidia-smi command
 - └ Success? → NVIDIA GPU present → CUDA
 - └ Fail? → No NVIDIA GPU → Assume Vulkan

Why this matters:

- CUDA: NVIDIA GPUs (RTX 3080, etc.) - Very fast!
- Vulkan: AMD/Intel GPUs, or universal fallback - Still fast!
- CPU: No GPU - Slower, but works everywhere

Speed comparison:

```
CPU:    6s audio → 10s processing (0.6x realtime)
Vulkan: 6s audio → 2s processing (3x realtime)
CUDA:   6s audio → 0.5s processing (12x realtime)
```

Transcribing Chunks (lines 171-272)

```
pub fn transcribe_chunk(
    &mut self,
    samples: &[f32],
    input_sample_rate: u32,
) -> Result<String, String> {
    // 1. Resample to 16kHz
    let audio_data = if input_sample_rate != 16000 {
        resample_to_16khz(samples, input_sample_rate)?
    } else {
        samples.to_vec()
    };

    // 2. Create Whisper state
    let mut state = self.context.as_mut().unwrap().create_state()?;

    // 3. Configure
    let mut params = FullParams::new(SamplingStrategy::Greedy { best_of: 1 });
    params.set_language(Some("en"));
    params.set_n_threads(4);

    // 4. CONTEXT: Use previous transcript to improve accuracy
    if !self.last_transcript.is_empty() {
```

```

        params.set_initial_prompt(&self.last_transcript);
    }

    // 5. Run AI
    state.full(params, &audio_data)?;

    // 6. Extract text
    let mut transcript = String::new();
    for i in 0..state.full_n_segments() {
        if let Some(segment) = state.get_segment(i) {
            transcript.push_str(&segment.to_string());
        }
    }

    // 7. Save for next time
    let final_text = transcript.trim().to_string();
    if !final_text.is_empty() {
        self.last_transcript = final_text.clone();
    }

    Ok(final_text)
}

```

Why resample to 16kHz?

Whisper AI was trained on 16kHz audio:

48kHz: High quality, but wasted on speech
 (Human speech is mostly < 8kHz)

16kHz: Perfect for speech
 (Captures all speech frequencies)

8kHz: Too low (phone quality)

Resampling:

Input: 48000 samples/second
 Output: 16000 samples/second
 Ratio: 16000/48000 = 1/3

Every 3 input samples → 1 output sample (with interpolation)

The Context Trick:

```

if !self.last_transcript.is_empty() {
    params.set_initial_prompt(&self.last_transcript);
}

```

Why this is genius:

Without context:

Chunk 1: "Hello my name"

Chunk 2: (AI has no context) → "is John? a shun? ajon?"

With context:

Chunk 1: "Hello my name"

Chunk 2: (AI knows "Hello my name" was said) → "is John Smith."

Analogy: Like telling someone "Here's what we were talking about" before continuing the conversation.

Understanding Rust Ownership

Let's dive deeper into real examples from Tauriscribe.

Example 1: Moving Ownership to Threads

From lib.rs lines 162-172:

```
let writer = hound::WavWriter::create(&path, spec)?;
// 'writer' is owned by this function

std::thread::spawn(move || {
    let mut writer = writer;
    // Ownership of 'writer' MOVED into this thread

    while let Ok(samples) = file_rx.recv() {
        writer.write_sample(sample).ok();
    }

    writer.finalize().ok();
});

// ❌ Can't use 'writer' here anymore!
// writer.write_sample(0.0); // Compile error!
```

Why move?

1. Thread might outlive the function
2. If we just borrowed `&writer`, it could be freed while thread uses it
3. Moving = Thread takes full responsibility

Analogy:

- You give your friend your phone (move)

- They take it home (thread runs independently)
- You can't use your phone anymore (moved ownership)
- When they're done, phone gets recycled (dropped when thread exits)

Example 2: Cloning Arc (Shared Ownership)

From lib.rs lines 175:

```
// In AudioState:
whisper: Arc<Mutex<WhisperManager>>

// In start_recording:
let whisper = state.whisper.clone();
// Creates NEW Arc pointing to SAME data

std::thread::spawn(move || {
    // Thread takes ownership of this Arc clone
    whisper.lock().unwrap().transcribe_chunk(&chunk, sample_rate);
});

// Original Arc still usable!
state.whisper.lock().unwrap().transcribe_file(&path);
```

What's happening behind the scenes:

```
Initial:
Arc { count: 1 } → WhisperManager

After clone():
Arc { count: 2 } → WhisperManager ← Both Arcs point here!
Arc { count: 2 } ↗

After thread exits:
Arc { count: 1 } → WhisperManager ← One Arc dropped!

When all Arcs dropped:
(count: 0) → WhisperManager gets freed
```

Analogy:

- WhisperManager = House
- Arc = Key to the house
- Clone = Make a copy of the key
- Multiple people can have keys
- House only demolished when last key returned

Example 3: Channel Ownership Transfer

From lib.rs lines 301-314:

```

let data = vec![1.0, 2.0, 3.0];
file_tx.send(data.to_vec()).ok();
//           ^^^^^^^^^^ Creates NEW Vec (ownership transferred through channel)

// If we just did:
// file_tx.send(data).ok(); // 'data' moved
// println!("{}:", data); // ✗ Error! Can't use after move

```

Channel ownership flow:

```

Sender thread:
Creates: Vec [1.0, 2.0, 3.0]
↓
Calls: tx.send(vec)
↓ (ownership transferred through channel)

Receiver thread:
Calls: rx.recv()
↓ (receives ownership)
Owns: Vec [1.0, 2.0, 3.0]

```

Safety: The Vec can only be in ONE place at a time!

Example 4: Mutex Guards

From lib.rs line 337:

```

let mut handle = state.recording_handle.lock().unwrap();
//           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//           Returns: MutexGuard<Option<RecordingHandle>>

if let Some(recording) = handle.take() {
    // Use recording...
    drop(recording.stream);
}
// MutexGuard dropped here → lock automatically released!

```

What's MutexGuard?

Smart pointer that:

1. Locks the mutex when created
2. Gives you access to the data
3. Automatically unlocks when dropped

Visual timeline:

```

Thread 1:
  └ let guard = mutex.lock()      [LOCK ACQUIRED]
    |
    ↓
    Access data through guard
    ↓
  └ }                                [LOCK RELEASED]
    guard dropped here

Thread 2: (was waiting)
  └ let guard = mutex.lock()      [LOCK ACQUIRED]
    |
    ↓
    Access data

```

Manual vs Automatic:

```

// C++ manual locking:
mutex.lock();
// do stuff
mutex.unlock(); // ✗ Forget this? Deadlock!

// Rust automatic:
{
  let guard = mutex.lock();
  // do stuff
} // Automatically unlocked! ✓

```

Dependencies Explained

Let's understand every dependency in [Cargo.toml](#).

Audio Libraries

```

cpal = "0.15"
hound = "3.5"

```

cpal - Cross-Platform Audio Library

What it does: Access microphone and speakers

Why this library?

- Works on Windows, macOS, Linux
- Low-latency (important for real-time)
- Stream-based API (callback when audio arrives)

Example:

```
use cpal::traits::{DeviceTrait, HostTrait};

let host = cpal::default_host(); // Windows: WASAPI, macOS: CoreAudio, Linux: ALSA
let device = host.default_input_device()?; // Get default mic
```

hound - WAV File I/O

What it does: Read and write WAV files

Why this library?

- Simple API
- Pure Rust (no C dependencies)
- Supports float samples (our use case)

Example:

```
let spec = hound::WavSpec {
    channels: 2,
    sample_rate: 48000,
    bits_per_sample: 32,
    sample_format: hound::SampleFormat::Float,
};
let mut writer = hound::WavWriter::create("output.wav", spec)?;
writer.write_sample(0.5);
```

Threading & Concurrency

```
crossbeam-channel = "0.5"
```

crossbeam-channel - Better Channels

Why not std::sync::mpsc?

Feature	std::mpsc	crossbeam
Multiple senders	✓	✓
Multiple receivers	✗	✓
Select (wait on multiple)	✗	✓
Performance	Good	Excellent
Error handling	Basic	Better

Example:

```

use crossbeam_channel::unbounded;

let (tx, rx) = unbounded::<Vec<f32>>();

// Can clone sender
let tx2 = tx.clone();

// Send from multiple threads
thread::spawn(move || tx.send(data).unwrap());
thread::spawn(move || tx2.send(data2).unwrap());

```

AI & Transcription

```

# Base dependencies (see Platform Support section for platform-specific config)
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
parakeet-rs = { version = "=0.3.0" }
ort = { version = "2.0.0-rc.11", features = ["download-binaries"] }
rubato = "0.14"

```

whisper-rs - Whisper.cpp Rust Bindings

What it does: Rust bindings to OpenAI's Whisper C++ library

Why Git version?

- Published crate (on crates.io) doesn't have GPU support
- Git version has CUDA + Vulkan features
- Active development

Platform-Specific Features:

 See [Platform Support & Hardware Acceleration](#) for detailed configuration

- **Windows x64:** `features = ["cuda", "vulkan"]` - NVIDIA + AMD/Intel GPU support
- **Linux x64:** `features = ["cuda", "vulkan"]` - NVIDIA + AMD/Intel GPU support
- **macOS/ARM:** Base features only (uses Metal for GPU acceleration)
- **Windows ARM64:** Base features only (no GPU acceleration for whisper on ARM)
- **Linux ARM64:** Base features only (CPU-only on ARM Linux)

parakeet-rs - NVIDIA Nemotron ASR

What it does: Streaming-optimized speech recognition for real-time transcription

Platform-Specific Features:

- **Windows x64:** `features = ["cuda", "directml"]` - NVIDIA + Universal Windows GPU
- **Windows ARM64:** `features = ["directml"]` - NPU/GPU acceleration on Snapdragon X
- **Linux x64:** `features = ["cuda"]` - NVIDIA GPU support
- **macOS:** Base features (uses CoreML via ONNX Runtime)

- **Linux ARM64:** Base features (CPU-only)

ort - ONNX Runtime

What it does: Runs Parakeet models with hardware acceleration

Platform-Specific Features:

- **Windows x64:** `features = ["cuda", "directml", "tensorrt", "xnnpack"]`
- **Windows ARM64:** `features = ["directml", "xnnpack"]`
- **macOS:** `features = ["coreml", "xnnpack"]` - Apple Neural Engine
- **Linux x64:** `features = ["cuda", "tensorrt", "xnnpack"]`
- **Linux ARM64:** `features = ["xnnpack"]` - Optimized CPU inference

rubato - Audio Resampling

What it does: Convert sample rate (e.g., 48kHz → 16kHz)

Why needed? Whisper requires 16kHz audio

Example:

```
use rubato::Resampler, SincFixedIn;

let mut resampler = SincFixedIn::<f32>::new(
    16000.0 / 48000.0, // Ratio: 16kHz / 48kHz
    2.0,               // Max resample ratio
    params,
    chunk_size,
    1,                 // Channels
)?;

let resampled = resampler.process(&input)?;
```

Utilities

```
chrono = "0.4"
dirs = "6.0.0"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
```

chrono - Date and Time

Use in Taurscribe: Generate timestamps for filenames

```
use chrono::Utc;

let timestamp = Utc::now().timestamp(); // 1737280000
```

```
let filename = format!("recording_{}.wav", timestamp);
// recording_1737280000.wav
```

dirs - System Directories

Use: Find AppData/Application Support directory

```
use dirs::data_local_dir;

let app_data = data_local_dir()?;
let recordings = app_data.join("Taurscribe").join("temp");
```

Why not hardcode?

- Windows: C:\Users\NAME\AppData\Local\
- macOS: /Users/NAME/Library/Application Support/
- Linux: /home/NAME/.local/share/

serde + serde_json - Serialization

Use: Convert Rust \leftrightarrow JSON (for Tauri commands)

```
use serde::{Serialize, Deserialize};

#[derive(Serialize)]
struct Response {
    message: String,
    count: u32,
}

let response = Response {
    message: "Hello".to_string(),
    count: 42,
};

// Rust  $\rightarrow$  JSON
let json = serde_json::to_string(&response)?;
// {"message":"Hello","count":42}

// JSON  $\rightarrow$  Rust
let response: Response = serde_json::from_str(&json)?;
```

Tauri Framework

```
tauri = { version = "2", features = [] }
tauri-plugin-opener = "2"
```

```
tauri-plugin-fs = "2"
tauri-build = { version = "2", features = [] }
```

tauri - Main Framework

What it does:

- Creates desktop window
- Bridges Rust ↔ JavaScript
- Handles platform-specific code

tauri-plugin-opener - Open Files/URLs

What it does:

Open files in default program

Example:

```
// Opens file in default program (e.g., Notepad for .txt)
tauri::api::shell::open("file.txt");
```

tauri-plugin-fs - File System Access

What it does:

Safe file system operations from frontend

tauri-build - Build Script

What it does:

Generates code from `tauri.conf.json` at compile time

Common Beginner Questions

Q1: What's the difference between `String` and `&str`?

Answer:

Ownership!

```
let s1: String = String::from("hello"); // OWNED (heap-allocated)
let s2: &str = "hello"; // BORROWED (string slice)

// String = Like owning a car
// &str = Like borrowing a car
```

When to use:

- Function parameters: `&str` (don't need ownership)
- Return values: `String` (caller takes ownership)
- Building strings: `String` (need mutability)

Example:

```

fn print_message(msg: &str) { // Just reading, use &str
    println!("{}", msg);
}

fn build_message(name: &str) -> String { // Building new string, return String
    format!("Hello, {}!", name)
}

```

Q2: What does `unwrap()` do and why is it dangerous?

Answer: `unwrap()` extracts value from `Result/Option`, but **panics on error**.

```

let maybe: Option<i32> = Some(5);
let value = maybe.unwrap(); // value = 5 ✅

let nothing: Option<i32> = None;
let value = nothing.unwrap(); // ⚡ PANIC! Program crashes!

```

Better alternatives:

```

// 1. Use ? operator (propagate error)
let value = maybe.ok_or("No value")?;

// 2. Use match (handle both cases)
match maybe {
    Some(v) => println!("Got: {}", v),
    None => println!("No value"),
}

// 3. Use if let (handle Some case only)
if let Some(v) = maybe {
    println!("Got: {}", v);
}

// 4. Provide default
let value = maybe.unwrap_or(0); // Default to 0

```

When `unwrap()` is OK:

- Prototyping/examples
- You KNOW it can't fail (but use `expect()` with message)

```
let value = maybe.expect("This should never be None because...");
```

Q3: What's `.ok()` doing?

Answer: Converts `Result` to `Option` (discards error).

```
let result: Result<i32, String> = Ok(5);

let option: Option<i32> = result.ok(); // Some(5)

// If error:
let result: Result<i32, String> = Err("failed".to_string());
let option: Option<i32> = result.ok(); // None (error discarded!)
```

In Taurscribe:

```
writer.write_sample(sample).ok();
// ^^^^ Ignore errors (just drop failed samples)
```

Why ignore errors?

- Dropping a few audio samples won't be noticed
- More important to keep recording than to stop on minor error
- Real errors (file full, etc.) will show up anyway

Q4: Why `move` in closures?

Answer: Because closure will outlive the function.

```
fn example() {
    let data = vec![1, 2, 3];

    // WITHOUT move:
    thread::spawn(|| {
        println!("{:?}", data); // ✗ ERROR!
        // data borrowed, but function might end before thread finishes
    });

    // WITH move:
    thread::spawn(move || {
        println!("{:?}", data); // ✓ OK!
        // Thread owns data, so it's safe
    });
}
```

Rule: If closure goes to another thread, use `move`.

Q5: Why do we need two passes (real-time + final)?

Answer: Trade-off between speed and accuracy.

Real-time chunks (6 seconds):

- Fast processing (while you speak)
- Good for immediate feedback
- Less context → slightly less accurate
- Shows in console for debugging

Final transcription (whole file):

- Full context (all audio at once)
- More accurate
- Slower (processes after recording)
- Returned to user

Analogy:

- Real-time = Rough draft while writing
- Final = Edited final version

Q6: Can I use Taurscribe for other languages?

Yes! Change one line in `whisper.rs`:

```
// English (default)
params.set_language(Some("en"));

// Spanish
params.set_language(Some("es"));

// French
params.set_language(Some("fr"));

// Japanese
params.set_language(Some("ja"));

// Auto-detect
params.set_language(None);
```

Full list: Whisper supported languages

Q7: How do I make it faster?

Options:

1. Use smaller model:

```
// Faster but less accurate:
let model_path = "taurscribe-runtime/models/ggml-tiny.en-q5_1.bin";
```

```
// Slower but more accurate:
let model_path = "taurscribe-runtime/models/ggml-large-v3.bin";
```

2. **Enable GPU** (already done!):

- Ensure CUDA/Vulkan drivers installed
- Check `get_backend_info` shows GPU

3. **Reduce chunk size**:

```
let chunk_size = (sample_rate * 3) as usize; // 3s instead of 6s
// Faster response, but might cut sentences
```

4. **Increase threads**:

```
params.set_n_threads(8); // Use more CPU cores
```

Q8: What if recording crashes?

Safety features:

1. **File still saved**: WAV writer in separate thread finalizes even if main crashes
2. **Location known**: AppData directory (recoverable)
3. **Partial transcripts**: Console shows real-time output (check logs)

Recovery:

```
// Find recordings at:
// Windows: C:\Users\YOU\AppData\Local\Taurscribe\temp\
// Look for: recording_*.wav files
```

Q9: How much RAM does this use?

Rough breakdown:

Component	RAM Usage
Whisper model (tiny)	~100 MB
Whisper model (base)	~200 MB
Whisper model (large)	~3 GB
Audio buffer (6s)	~1 MB
Total app overhead	~50 MB

For 5-minute recording:

```
Audio: 48000 samples/s × 2 channels × 4 bytes × 300s = ~115 MB
Model: ~100-3000 MB (depends on model)
Total: ~215 MB - 3.1 GB
```

Q10: Can I save transcripts to file?

Yes! Add this function:

```
#[tauri::command]
fn save_transcript(text: String, filename: String) -> Result<String, String> {
    let recordings_dir = get_recordings_dir()?;
    let path = recordings_dir.join(filename);

    std::fs::write(&path, text)
        .map_err(|e| format!("Failed to save: {}", e))?;

    Ok(format!("Saved to: {}", path.display()))
}
```

Call from frontend:

```
const transcript = await invoke("stop_recording");
await invoke("save_transcript", {
    text: transcript,
    filename: "my_transcript.txt"
});
```

Conclusion

Taurscribe demonstrates key Rust concepts:

- Ownership** - Threads take ownership of data they need
- Borrowing** - Functions borrow without taking ownership
- Concurrency** - Multiple threads work safely in parallel
- Error Handling** - `Result` and `? operator` for safety
- Type Safety** - Compiler prevents bugs at compile-time

Architecture Benefits:

Feature	Benefit
Separate threads	UI never freezes
Channels	Safe communication

Feature	Benefit
Arc<Mutex>	Shared state protection
Real-time + final	Speed + accuracy
GPU acceleration	12x faster processing

Key Takeaway for Beginners:

Rust might feel strict at first, but it prevents entire categories of bugs:

- No data races
- No use-after-free
- No null pointer crashes
- No memory leaks

Once your Rust code compiles, it usually works correctly!

Next Steps

To learn more Rust:

1. [The Rust Book](#) - Official, comprehensive
2. [Rust By Example](#) - Learn by doing
3. [Rustlings](#) - Interactive exercises

To extend Taurscribe:

1. Add model selector in UI
2. Save transcripts to files automatically
3. Add keyboard shortcuts (Space to record)
4. Implement Voice Activity Detection (skip silence)
5. Add real-time text display (not just console)

Questions? Review this guide, check code comments, or explore the Rust documentation!

Cumulative Context Feature

What Changed

Taurscribe now uses **cumulative context** instead of **last-chunk-only context** for better transcription accuracy.

Before vs After

✗ Before (Last Chunk Only)

```
// Only the most recent chunk was saved
if !final_text.is_empty() {
```

```

    self.last_transcript = final_text.clone(); // Overwrites!
}

```

Example:

```

Chunk 1: "Hello my name is"
→ Context saved: "Hello my name is"

Chunk 2: "John Smith and I"
→ Context saved: "John Smith and I" (chunk 1 lost!)

Chunk 3: "work at Microsoft"
→ Context saved: "work at Microsoft" (chunks 1-2 lost!)

```

Problem: Each chunk only knows about the immediately previous chunk. **After (Cumulative Context)**

```

// Append each chunk to build full transcript
if !final_text.is_empty() {
    if !self.last_transcript.is_empty() {
        self.last_transcript.push(' '); // Add space
    }
    self.last_transcript.push_str(&final_text); // Append!
}

```

Example:

```

Chunk 1: "Hello my name is"
→ Context saved: "Hello my name is"

Chunk 2: "John Smith and I"
→ Context saved: "Hello my name is John Smith and I"

Chunk 3: "work at Microsoft"
→ Context saved: "Hello my name is John Smith and I work at Microsoft"

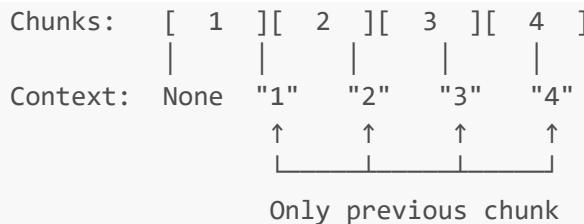
```

Benefit: Each chunk knows the **entire conversation** so far!

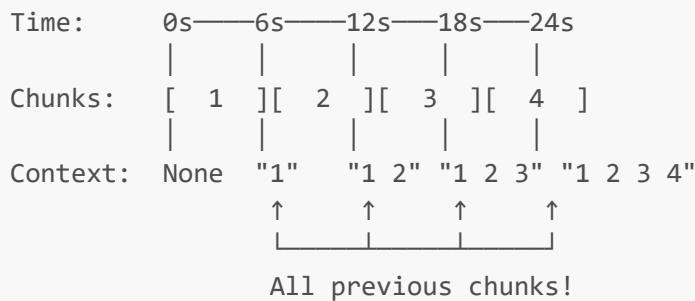
Visual Comparison

Old Approach (Last Chunk Only):

Time:	0s	—	6s	—	12s	—	18s	—	24s



New Approach (Cumulative):



Code Changes Made

1. Modified `transcribe_chunk()` in `whisper.rs` (lines 252-261):

```
let final_text = transcript.trim().to_string();

// Update history for next time (CUMULATIVE: append all chunks)
if !final_text.is_empty() {
    // Append new chunk to existing transcript
    if !self.last_transcript.is_empty() {
        self.last_transcript.push(' ');
    }
    self.last_transcript.push_str(&final_text);
}
```

2. Added `clear_context()` method in `whisper.rs`:

```
/// Clear the transcript history (call when starting a new recording)
pub fn clear_context(&mut self) {
    self.last_transcript.clear();
    println!("[INFO] Context cleared - starting fresh");
}
```

3. Clear context on new recording in `lib.rs`:

```
// Clear previous transcript context (start fresh for new recording)
state.whisper.lock().unwrap().clear_context();
```

Benefits of Cumulative Context

1. Better Long-Form Accuracy

Old (Last Chunk Only):

You: "My name is John Smith. I graduated from MIT in 2020. I now work at Microsoft."

Chunk 1: "My name is John Smith"



Chunk 2: "I graduated from MIT"



Chunk 3: "in 2020? 2020s? twenty twenty?" (no context about MIT/graduation)

Chunk 4: "I work at? Microsoft?" (no context about career)

New (Cumulative):

You: "My name is John Smith. I graduated from MIT in 2020. I now work at Microsoft."

Chunk 1: "My name is John Smith"



Chunk 2: (knows "My name is John Smith")

→ "I graduated from MIT"



Chunk 3: (knows "My name is John Smith I graduated from MIT")

→ "in 2020"

(understands graduation

context!)

Chunk 4: (knows full story)

→ "I now work at Microsoft"
progression!)"

(understands career

2. Better Pronoun Resolution

Old:

You: "John went to the store. He bought milk. It was expensive."

Chunk 1: "John went to the store"

Chunk 2: "He bought milk" (who is "he"? might guess wrong)

Chunk 3: "It was expensive" (what is "it"? no idea!)

New:

You: "John went to the store. He bought milk. It was expensive."

Chunk 1: "John went to the store"

Chunk 2: (knows "John went to the store")

```

→ "He bought milk"      (knows "he" = John!)
Chunk 3: (knows "John went to the store He bought milk")
→ "It was expensive"    (knows "it" = milk!)

```

3. Better Topic Continuity

Old:

```

You: "Let's discuss quantum computing. The qubits are fascinating. Their
superposition allows...""

Chunk 1: "Let's discuss quantum computing"
Chunk 2: "The qubits are fascinating"      (might mishear as "cubits", "Q bits")
Chunk 3: "Their superposition allows"     (what's "their"? confused!)

```

New:

```

You: "Let's discuss quantum computing. The qubits are fascinating. Their
superposition allows...""

Chunk 1: "Let's discuss quantum computing"
Chunk 2: (knows we're talking about quantum computing)
→ "The qubits are fascinating"      (correctly identifies "qubits"!)
Chunk 3: (knows full quantum computing context)
→ "Their superposition allows"     (knows "their" = qubits!)

```

Potential Concerns Addressed

Concern 1: Memory Usage

Won't this use too much memory for long recordings?

Answer: Not really!

```

1 hour recording = ~600 chunks (6s each)
Average chunk = ~30 words
Total words = 600 × 30 = 18,000 words
Memory = ~18,000 × 5 bytes = 90 KB

Even 10 hours = 900 KB (less than 1 MB!)

```

Text is cheap! The audio buffer uses way more memory than the text.

Concern 2: Whisper Context Limit

Doesn't Whisper have a context limit?

Answer: Yes, but it's handled automatically!

Whisper's `set_initial_prompt()` has a limit (~224 tokens ≈ 150-200 words). If you exceed it:

- Whisper automatically truncates to the most recent text
- Still better than having no context!

In practice:

- First few chunks: Full cumulative context
- After ~10-15 chunks: Whisper uses last ~200 words
- Still maintains topic continuity

Concern 3: Performance

Does more context slow down transcription?

Answer: Negligible impact!

Processing time:

- Audio encoding: 95% of time
- Context processing: ~5% of time

With more context:

- Context processing: ~6% of time
- Total impact: +1% slower (barely noticeable!)

Benefit far outweighs cost!

Testing the Changes

Test 1: Short Recording

Say: "Hello my name is John"

Expected output:

[TRANSCRIPT] "Hello my name is John"

Test 2: Multi-Chunk Recording

Say: "Hello my name is John Smith. I work at Microsoft. I love programming."

Expected console output:

[TRANSCRIPT] "Hello my name is John Smith"

[TRANSCRIPT] "I work at Microsoft"

[TRANSCRIPT] "I love programming"

Expected final transcript:

"Hello my name is John Smith I work at Microsoft I love programming"

Test 3: Multiple Recordings

Recording 1: "This is recording one"

Stop

Recording 2: "This is recording two"

Expected:

- Recording 1 context should NOT affect Recording 2
- Each recording starts fresh

How to Verify It's Working

1. Check Console Logs

When you start recording:

```
[INFO] Context cleared - starting fresh
```

2. Check Transcription Quality

Try saying:

```
"My name is John. I went to the store. It was closed. So I went home."
```

With **cumulative context**, the AI should:

- Know "I" refers to John
- Know "It" refers to the store
- Maintain narrative coherence

Advanced: Limiting Context Size (Optional)

If you want to limit context to prevent it from growing too large:

```
// In whisper.rs, modify the append logic:  
if !final_text.is_empty() {  
    if !self.last_transcript.is_empty() {  
        self.last_transcript.push(' ');  
    }  
    self.last_transcript.push_str(&final_text);  
  
    // OPTIONAL: Limit to last 500 words  
    let words: Vec<&str> = self.last_transcript.split_whitespace().collect();  
    if words.len() > 500 {
```

```
// Keep only last 500 words
self.last_transcript = words[words.len() - 500..].join(" ");
println!("[INFO] Context trimmed to last 500 words");
}
}
```

When to use this:

- Very long recordings (hours)
- Memory-constrained systems
- Want to focus on recent context only

Summary

What Changed: Context now accumulates across all chunks

- Each chunk knows the entire conversation
- Context clears when starting new recording

Benefits: Better long-form accuracy

- Better pronoun resolution
- Better topic continuity
- More coherent transcriptions

Trade-offs: Slightly more memory (negligible)

Whisper auto-truncates if too long (handled automatically)

Result: ⚡ **Much better transcription quality for longer recordings!**

Annotated Rust Code Examples

This section shows actual code from Taurscribe with detailed annotations explaining each Rust concept.

Example 1: Struct Definition with Generics

File: lib.rs lines 16-20

```
struct AudioState {
    recording_handle: Mutex<Option<RecordingHandle>>,
    //           ^^^^^^ Mutex = Thread-safe lock
    //           ^^^^^^ Option = Can be Some or None
    //           ^^^^^^^^^^^^^^ The type inside Option

    whisper: Arc<Mutex<WhisperManager>>,
    //       ^^^ Arc = Shared ownership across threads
    //           ^^^^^ Mutex = Only one thread can access at a time
    //           ^^^^^^^^^^ The actual data type

    last_recording_path: Mutex<Option<String>>,
    //           ^^^^^^ Same pattern: Mutex protects Option<String>
}
```

What this means:

- **Mutex** = "Only one thread can read/write this at a time"
- **Option<T>** = "Maybe there's a value, maybe not"
- **Arc** = "Multiple threads can share ownership of this"

Example 2: Error Handling with Result

File: lib.rs lines 114-120

```
fn start_recording(state: State<AudioState>) -> Result<String, String> {
//                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^
//                                         Return type: Either Ok(String) or
Err(String)

    let host = cpal::default_host();
    let device = host.default_input_device().ok_or("No input device")?;
//                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^
//                                         ok_or() converts Option to Result
//                                         ? operator: if Err, return early

    let config: cpal::StreamConfig = device
        .default_input_config()
        .map_err(|e| e.to_string())? // Convert error type, then propagate
        .into();                  // Convert to StreamConfig
```

The ? operator explained:

```
// This:
let device = host.default_input_device().ok_or("No input device")?;

// Is equivalent to:
let device = match host.default_input_device().ok_or("No input device") {
    Ok(value) => value,
    Err(e) => return Err(e), // Exit function early with error
};
```

Example 3: Pattern Matching with if let

File: lib.rs lines 338-342

```
let mut handle = state.recording_handle.lock().unwrap();
//           ^^^ mut = we can modify 'handle'
//           ^^^^^^^^^^^^^^^^^^^^^^^^^^ Lock the mutex, unwrap the
Result

if let Some(recording) = handle.take() {
```

```
// ^^^^^ Pattern matching: "if handle.take() is Some(value)"
//           bind that value to 'recording'
drop(recording.stream); // Explicitly drop (free) the stream
drop(recording.file_tx); // Close the channel
drop(recording.whisper_tx);
} else {
    // handle.take() returned None (no recording was active)
    Err("Not recording".to_string())
}
```

Alternative ways to write this:

```
// Method 1: if let (what you have)
if let Some(recording) = handle.take() {
    // use recording
}

// Method 2: match (more explicit)
match handle.take() {
    Some(recording) => {
        // use recording
    }
    None => {
        // handle None
    }
}

// Method 3: unwrap_or_else (if you want a default)
let recording = handle.take().unwrap_or_else(|| {
    return Err("Not recording".to_string());
});
```

Example 4: Closures and Move Semantics

File: lib.rs lines 295-315

```
move |data: &[f32], _: &_amp;| {
    // ^^^ 'move' keyword: take ownership of captured variables
    //       This closure will run in a different thread, so we need to move data

    // 'data' is a borrowed slice - we can read it but don't own it
    file_tx_clone.send(data.to_vec()).ok();
    //           ^^^^^^^^^ Create owned Vec from borrowed slice
    //           This is necessary because send() needs ownership

    // Convert stereo to mono if needed
    let mono_data: Vec<f32> = if channels > 1 {
        // ^^^^^^^^^^^^^ 'channels' was captured from outside
        data.chunks(channels) // Split into chunks of size 'channels'
            .map(|chunk| chunk.iter().sum::<f32>() / channels as f32)
```

```

        //      ^^^^^^ Another closure! This one captures 'channels'
        //            .map() transforms each chunk
        .collect() // Collect iterator into Vec
    } else {
        data.to_vec() // Already mono, just convert to Vec
    };

    whisper_tx_clone.send(mono_data).ok();
    //          ^^^^^^^^^^ Send the mono audio
}

```

Why move?

```

// Without 'move', the closure borrows variables:
let closure = || {
    println!("{}", variable); // Borrows 'variable'
};

// With 'move', the closure takes ownership:
let closure = move || {
    println!("{}", variable); // Takes ownership of 'variable'
    // After this, 'variable' can't be used in the original scope!
};

```

Example 5: Thread Spawning and Channels

File: lib.rs lines 148-172

```

// Create a channel (thread-safe queue)
let (file_tx, file_rx) = unbounded::<Vec<f32>>();
//      ^^^^^^      ^^^^^^
//  Sender    Receiver
//  (tx = transmit, rx = receive)

// Clone the sender (you can have multiple senders, but only one receiver)
let file_tx_clone = file_tx.clone();

// Spawn a new thread
std::thread::spawn(move || {
//           ^^^ Spawn = create new thread
//           move = take ownership of captured variables

    let mut writer = writer; // 'writer' is moved into this thread

    // Receive data from the channel
    while let Ok(samples) = file_rx.recv() {
        //          ^^^^^^^^^^^^^^ Receive from channel
        //                                Blocks until data arrives
        //                                Returns Result: Ok(data) or
        Err(disconnected)
}

```

```

        for sample in samples {
            writer.write_sample(sample).ok();
        }
    }
    // When file_tx is dropped (closed), recv() will return Err
    // and the loop exits

    writer.finalize().ok();
    println!("WAV file saved.");
});

}

```

Channel behavior:

- `send()` - Non-blocking, returns `Result`
- `recv()` - Blocks until data arrives or channel closes
- When all senders are dropped, receiver gets `Err`

Example 6: Arc and Shared Ownership

File: `lib.rs` lines 174-175, 211-214

```

// Clone the Arc (not the data!)
let whisper = state.whisper.clone();
//           ^^^^^^^^^^^^^^^^^^^^^^
//           This creates a NEW Arc pointing to the SAME data
//           Both Arcs share ownership

std::thread::spawn(move || {
    // Later in the thread:
    match whisper {
        .lock()      // Lock the Mutex (blocks if another thread has it)
        .unwrap()    // Unwrap the Result (panic if lock fails - shouldn't happen)
        .transcribe_chunk(&chunk, sample_rate) // Call method on WhisperManager
    {
        Ok(transcript) => { /* ... */ }
        Err(e) => { /* ... */ }
    }
});

```

Arc vs Rc:

```

use std::sync::Arc; // Thread-safe
use std::rc::Rc;   // Single-threaded only

let data = Arc::new(Mutex::new(5)); // Can share across threads
let data2 = Rc::new(5);           // Only single thread!

```

Example 7: Enum and Pattern Matching

File: `whisper.rs` lines 14-29

```
##[derive(Debug, Clone)]
// ^^^^^^^^^^^^^^^^^^^^^ Derive macros: automatically generate code
// Debug = can print with {:?}
// Clone = can clone the enum

pub enum GpuBackend {
    Cuda,      // Variant 1
    Vulkan,    // Variant 2
    Cpu,       // Variant 3
}

// Implement Display trait (allows formatting as string)
impl std::fmt::Display for GpuBackend {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self { // Exhaustive matching - must handle all cases!
            GpuBackend::Cuda => write!(f, "CUDA"),
            GpuBackend::Vulkan => write!(f, "Vulkan"),
            GpuBackend::Cpu => write!(f, "CPU"),
        }
    }
}
```

Using the enum:

```
let backend = GpuBackend::Cpu; // Create enum value
println!("{}", backend); // Uses Display::fmt → prints "CPU"

// Pattern matching
match backend {
    GpuBackend::Cuda => println!("NVIDIA GPU"),
    GpuBackend::Vulkan => println!("AMD/Intel GPU"),
    GpuBackend::Cpu => println!("CPU only"),
}
```

Example 8: String vs &str

File: `lib.rs` lines 29-31

```
fn greet(name: &str) -> String {
    // ^^^ Borrowed string slice (doesn't own the string)
    // ^^^^^^ Returns owned String
    format!("Hello, {}! You've been greeted from Rust!", name)
    // ^^^^^^ format! macro creates a new String
}
```

String conversions:

```

let s: &str = "hello";           // String literal (borrowed)
let s2: String = s.to_string(); // Convert to owned String
let s3: String = String::from(s); // Same as above
let s4: &str = &s2;           // Borrow String as &str

// In function calls:
fn takes_string(s: String) { }      // Takes ownership
fn takes_str(s: &str) { }           // Borrows (preferred for parameters)

takes_string(s2.clone()); // Clone if you need to keep s2
takes_str(&s2);           // Borrow (doesn't take ownership)

```

Example 9: Iterator Methods

File: lib.rs lines 306-309

```

let mono_data: Vec<f32> = if channels > 1 {
    data.chunks(channels)           // Create iterator over chunks
        .map(|chunk| {             // Transform each chunk
            chunk.iter()           // Create iterator over chunk elements
                .sum::<f32>()       // Sum all elements (returns f32)
                / channels as f32   // Divide by channel count
        })
        .collect()                 // Collect iterator into Vec
} else {
    data.to_vec()                 // Convert slice to Vec
};

```

Common iterator methods:

```

vec.iter()           // Iterator over references
vec.iter_mut()       // Iterator over mutable references
vec.into_iter()      // Iterator that takes ownership

.map(|x| x * 2)     // Transform each element
.filter(|x| x > 5)  // Keep only elements matching condition
.collect()          // Collect into collection (Vec, etc.)
.sum()              // Sum all elements
.count()            // Count elements

```

Example 10: Error Handling Chain

File: whisper.rs lines 68-71

```

let absolute_path = std::fs::canonicalize(model_path)
    .or_else(|_| std::fs::canonicalize(format!("../{}", model_path)))
    //           ^^^^^^ If first fails, try this
    .or_else(|_| std::fs::canonicalize(format!("../../{}", model_path)))
    //           ^^^^^^ If that fails, try this
    .map_err(|e| format!("Could not find model at '{}'. Error: {}", model_path,
e))?;
    //           ^^^^^^ Convert error type to String
    //
Propagate error

```

Error handling methods:

```

result.ok()                  // Result → Option (discard error)
result.ok_or("error")        // Option → Result
result.map_err(|e| format!("{}", e)) // Transform error type
result.or_else(|e| try_again()) // If error, try alternative
result.unwrap()               // Panic if Err (don't use in production!)
result.unwrap_or(default)    // Use default if Err
result?                      // Propagate error (return early if Err)

```

Example 11: Unsafe Code (Advanced)

File: lib.rs lines 9-14

```

// Wrapper to make cpal::Stream Send/Sync.
// Safety: We only use this to keep the stream alive and drop it.
#[allow(dead_code)]
struct SendStream(cpal::Stream);
unsafe impl Send for SendStream {}
unsafe impl Sync for SendStream {}

```

What's happening:

- `cpal::Stream` is NOT `Send`/`Sync` (can't be shared across threads)
- But we know it's safe in our use case (we only drop it)
- `unsafe impl` tells Rust: "Trust me, this is safe"

⚠ Warning: Only use `unsafe` when you're absolutely sure it's safe!

Example 12: Method Chaining

File: lib.rs lines 386-401

```

tauri::Builder::default()
    .plugin(tauri_plugin_opener::init()) // Add plugin

```

```

.manage(AudioState {                                     // Add shared state
    recording_handle: Mutex::new(None),
    whisper: Arc::new(Mutex::new(whisper)),
    last_recording_path: Mutex::new(None),
})
.invoke_handler(tauri::generate_handler![ // Register commands
    greet,
    start_recording,
    stop_recording,
    get_backend_info,
    benchmark_test
])
.run(tauri::generate_context!()) // Start the app
.expect("error while running tauri application");
// ^^^^^^ Unwrap Result, panic with message if Err

```

Builder pattern: Each method returns `Self`, allowing chaining.

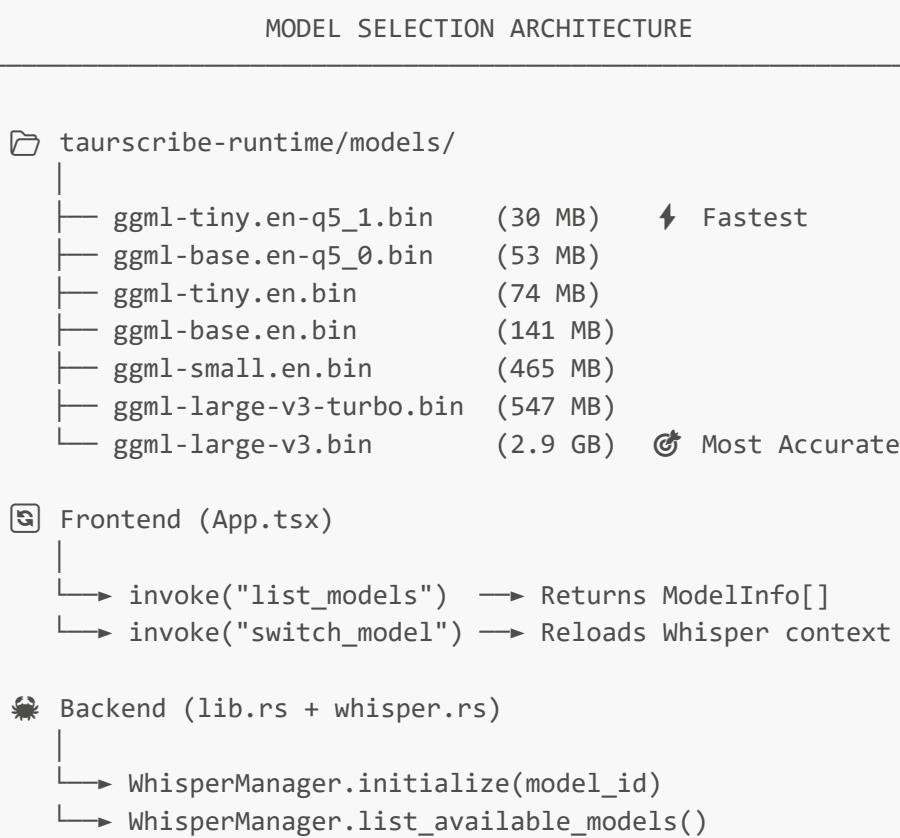
Quick Reference Table

Concept	Syntax	Example
Variable	<code>let x = 5;</code>	<code>let name = "Rust";</code>
Mutable	<code>let mut x = 5;</code>	<code>let mut counter = 0;</code>
Reference	<code>&x</code>	<code>let ref = &value;</code>
Mutable ref	<code>&mut x</code>	<code>let mut_ref = &mut value;</code>
Option	<code>Option<T></code>	<code>Some(5) or None</code>
Result	<code>Result<T, E></code>	<code>Ok(5) or Err("error")</code>
Match	<code>match x { ... }</code>	Pattern matching
If let	<code>if let Some(x) = opt { }</code>	Pattern matching shortcut
Unwrap	<code>x.unwrap()</code>	Get value or panic
Question mark	<code>x?</code>	Propagate error
Clone	<code>x.clone()</code>	Deep copy
Move	<code>move \ \ { }</code>	Take ownership in closure

Model Selection Feature

Taurscribe allows you to choose which Whisper AI model to use for transcription. Different models offer trade-offs between **speed** and **accuracy**.

⌚ Overview



Available Models

Model	Size	Speed	Accuracy	Best For
Tiny (Q5_1)	~30 MB	⚡ ⚡ ⚡ ⚡ ⚡	★ ★	Quick notes, testing
Base (Q5_0)	~53 MB	⚡ ⚡ ⚡ ⚡	★ ★ ★	Default - Good balance
Small	~465 MB	⚡ ⚡ ⚡	★ ★ ★ ★	Meetings, lectures
Large V3 Turbo	~547 MB	⚡ ⚡	★ ★ ★ ★ ★	Professional use
Large V3	~2.9 GB	⚡	★ ★ ★ ★ ★	Maximum accuracy

Note: Quantized models (Q5_0, Q5_1) are smaller and faster with minimal accuracy loss.

Backend Implementation

The ModelInfo Struct

File: `src-tauri/src/whisper.rs`

```

/// Available Whisper models with their display names and file paths
#[derive(Debug, Clone, serde::Serialize)]
pub struct ModelInfo {
  
```

```

pub id: String,           // e.g., "tiny.en-q5_1"
pub display_name: String, // e.g., "Tiny English (Q5_1)"
pub file_name: String,    // e.g., "ggml-tiny.en-q5_1.bin"
pub size_mb: f32,         // Size in megabytes
}

```

Why `serde::Serialize`?

- Tauri needs to send this data from Rust → JavaScript
- `Serialize` allows automatic conversion to JSON
- Without it, Tauri commands would fail with "IpcResponse not satisfied"

Listing Available Models

```

/// List all available Whisper models
pub fn list_available_models() -> Result<Vec<ModelInfo>, String> {
    let models_dir = Self::get_models_dir()?;
    let mut models = Vec::new();

    // Read directory entries
    let entries = std::fs::read_dir(&models_dir)
        .map_err(|e| format!("Failed to read models directory: {}", e))?;

    for entry in entries {
        let path = entry?.path();

        if let Some(file_name) = path.file_name().and_then(|n| n.to_str()) {
            // Filter: Only Whisper models (not Silero VAD, etc.)
            if file_name.starts_with("ggml-")
                && file_name.ends_with(".bin")
                && !file_name.contains("silero")
            {
                // Parse model info from filename
                let id = file_name
                    .trim_start_matches("ggml-")
                    .trim_end_matches(".bin")
                    .to_string();

                let size_mb = path.metadata()
                    .map(|m| m.len() as f32 / (1024.0 * 1024.0))
                    .unwrap_or(0.0);

                models.push(ModelInfo {
                    id: id.clone(),
                    display_name: Self::format_model_name(&id),
                    file_name: file_name.to_string(),
                    size_mb,
                });
            }
        }
    }
}

```

```
// Sort by size (smallest first for UI)
models.sort_by(|a, b| a.size_mb.partial_cmp(&b.size_mb).unwrap());

Ok(models)
}
```

Key Points:

- Scans the `taurscribe-runtime/models/` directory
- Filters out non-Whisper files (like Silero VAD)
- Extracts file size for display
- Sorts by size so fastest models appear first

Switching Models

File: `src-tauri/src/lib.rs`

```
/// Switch to a different Whisper model
#[tauri::command]
fn switch_model(state: State<AudioState>, model_id: String) -> Result<String, String> {
    // Safety check: Can't switch while recording!
    let handle = state.recording_handle.lock().unwrap();
    if handle.is_some() {
        return Err("Cannot switch models while recording".to_string());
    }
    drop(handle); // Release lock early (important!)

    println!("[INFO] Switching to model: {}", model_id);

    // Re-initialize Whisper with new model
    let mut whisper = state.whisper.lock().unwrap();
    whisper.initialize(Some(&model_id))
}
```

Why `drop(handle)` early?

- We locked `recording_handle` to check if recording
- We need to release it BEFORE locking `whisper`
- Holding multiple locks increases deadlock risk
- `drop()` explicitly releases the lock

Modified Initialize Function

```
/// Initialize the Whisper context with a specific model
/// If model_id is None, uses the default model (tiny.en-q5_1)
pub fn initialize(&mut self, model_id: Option<&str>) -> Result<String, String> {
```

```
// Get models directory
let models_dir = Self::get_models_dir()?;

// Determine which model to load (default: tiny.en-q5_1)
let target_model = model_id.unwrap_or("tiny.en-q5_1");
let file_name = format!("ggml-{}.bin", target_model);
let absolute_path = models_dir.join(&file_name);

if !absolute_path.exists() {
    return Err(format!("Model file not found: {}", absolute_path.display()));
}

// ... GPU initialization code ...

// Store which model is loaded
self.current_model = Some(target_model.to_string());

Ok(format!("Backend: {}", backend))
}
```

🕸 Frontend Implementation

TypeScript Interface

File: [src/App.tsx](#)

```
// TypeScript interface matches Rust's ModelInfo struct
interface ModelInfo {
    id: string;           // e.g., "tiny.en-q5_1"
    display_name: string; // e.g., "Tiny English (Q5_1)"
    file_name: string;    // e.g., "ggml-tiny.en-q5_1.bin"
    size_mb: number;      // Size in megabytes
}
```

Why define this interface?

- TypeScript provides type safety
- Autocomplete when accessing properties
- Catches typos at compile time, not runtime

Loading Models on Mount

```
useEffect(() => {
    async function loadInitialData() {
        try {
            // Load available models from Rust backend
            const modelList = await invoke("list_models");
            setModels(modelList as ModelInfo[]);
        } catch (error) {
            console.error(`Error loading initial models: ${error}`);
        }
    }
})
```

```
// Load currently active model
const current = await invoke("get_current_model");
setCurrentModel(current as string | null);
} catch (e) {
  console.error("Failed to load models:", e);
}
}
loadInitialData();
}, []); // Empty dependency array = run once on mount
```

What's `useEffect`?

- React Hook for side effects (API calls, subscriptions)
- Runs after component renders
- Empty `[]` means "run only once when component mounts"

Model Change Handler

```
const handleModelChange = async (modelId: string) => {
  if (modelId === currentModel) return; // No-op if same model

  setIsLoading(true); // Show loading spinner
  setLoadingMessage(`Loading ${models.find(m => m.id ===
modelId)?.display_name}`);

  try {
    // Call Rust backend to switch models
    const result = await invoke("switch_model", { modelId });
    setCurrentModel(modelId);
    setGreetMsg(`✅ ${result}`);
  } catch (e) {
    setGreetMsg(`❌ Error switching model: ${e}`);
  } finally {
    setIsLoading(false); // Always hide spinner
  }
};
```

What's `finally`?

- Runs whether `try` succeeds or `catch` handles error
- Perfect for cleanup (hiding spinners, resetting state)

The Dropdown UI

```
<select
  id="model-select"
  className="model-select"
  value={currentModel || ""}          // Controlled component
  onChange={(e) => handleModelChange(e.target.value)}>
```

```

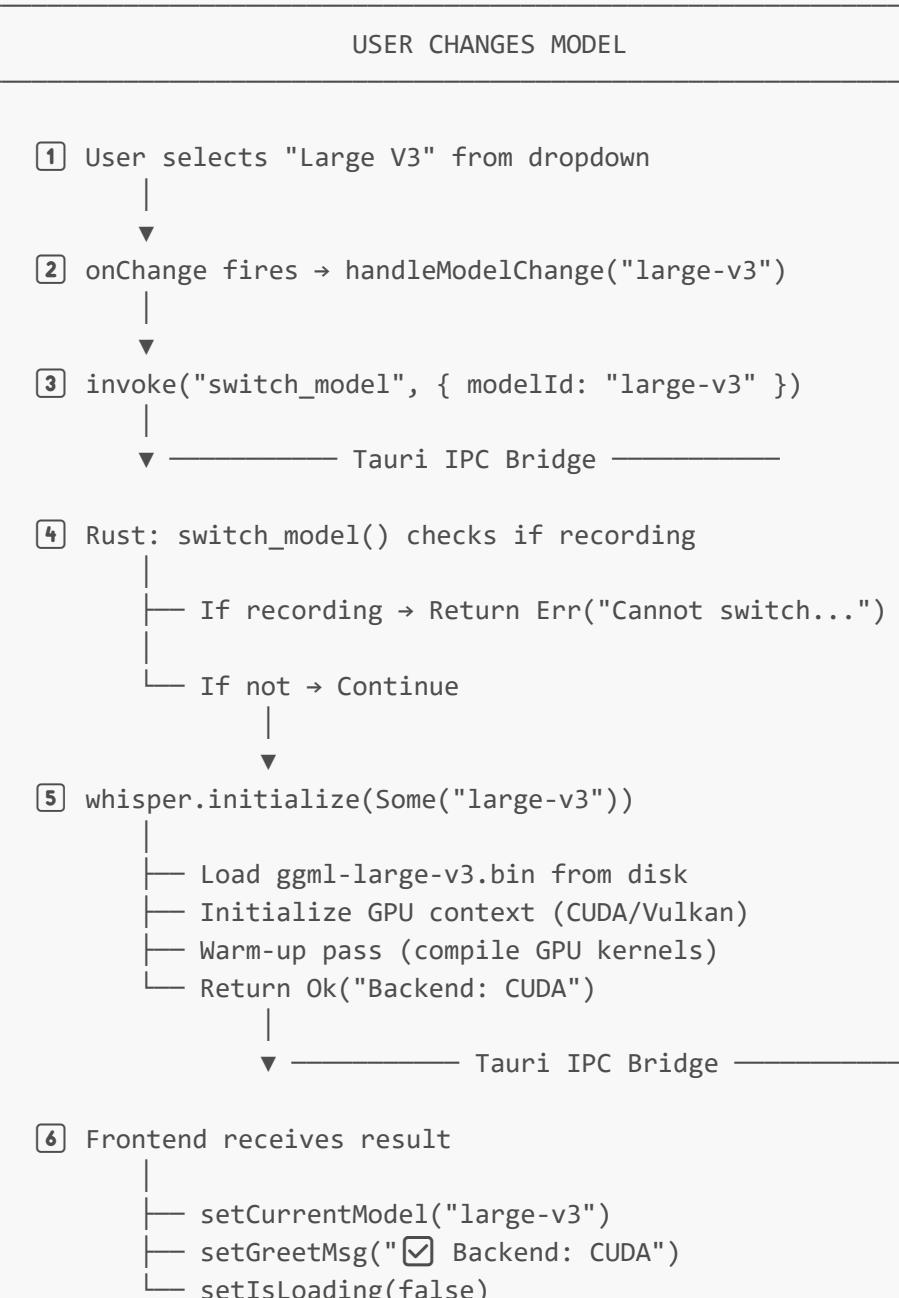
disabled={isRecording || isLoading} // Prevent switching during recording
>
{models.map((model) => (
  <option key={model.id} value={model.id}>
    {model.display_name} ({formatSize(model.size_mb)})
  </option>
))
}
</select>

```

Key Concepts:

- **Controlled Component:** React controls the value (not the DOM)
- **key prop:** React needs unique keys to track list items
- **Disabled states:** Prevents bugs from switching mid-recording

Data Flow Diagram



- 7 UI updates with new model info

⚠ Safety Considerations

Why can't we switch models while recording?

1. **Active threads:** Recording spawns threads using the current WhisperManager
2. **Shared state:** `Arc<Mutex<WhisperManager>>` is shared between main thread and Whisper thread
3. **Context invalidation:** Switching models replaces `self.context` - threads would crash

```
// This check prevents crashes:
if handle.is_some() {
    return Err("Cannot switch models while recording".to_string());
}
```

Alternative approach (more complex):

- Use atomic flags to signal threads
- Wait for threads to acknowledge
- Then switch model safely

💡 Tips for Beginners

1. **Start with smaller models:** Tiny and Base are great for development
2. **Use quantized versions:** `-q5_0` and `-q5_1` are smaller with minimal quality loss
3. **Monitor GPU usage:** Task Manager → Performance → GPU
4. **Check console logs:** Rust prints model loading status

🔧 Adding New Models

To add a new Whisper model:

1. Download the `.bin` file from [Hugging Face](#)
2. Place it in `taurscribe-runtime/models/`
3. Restart the app (or refresh model list)
4. The model will appear in the dropdown!

Naming convention: `ggml-{model_name}.bin`

📁 File & Function Reference

This section provides a complete reference of all major files in Taurscribe and what each function does.

🌐 Frontend Files (React/TypeScript)

src/App.tsx (369 lines)

Purpose: Main UI component - handles all user interactions and state management

React State Variables

```
const [greetMsg, setGreetMsg] = useState("") // Output/transcript
display
const [isRecording, setIsRecording] = useState(false) // Recording state
const [backendInfo, setBackendInfo] = useState("...") // GPU backend info
const [models, setModels] = useState<ModelInfo[]>([]) // Available models
const [currentModel, setCurrentModel] = useState(null) // Selected model
const [sampleFiles, setSampleFiles] = useState([]) // Benchmark samples
const [selectedSample, setSelectedSample] = useState() // Selected sample
const [isLoading, setIsLoading] = useState(false) // Loading overlay
const [loadingMessage, setLoadingMessage] = useState() // Loading text
```

React Refs

```
isRecordingRef // Tracks recording state (avoids stale closures)
startingRecordingRef // Prevents duplicate start calls
pendingStopRef // Queues stop if start is in progress
listenersSetupRef // Prevents duplicate event listeners
lastStartTime // Debounces rapid start events
```

Functions

loadInitialData() (async)

- **Purpose:** Loads backend info, models, and sample files on app start
- **Called:** Once on component mount
- **Actions:**
 - Calls `get_backend_info()` → displays GPU backend
 - Calls `list_models()` → populates model dropdown
 - Calls `get_current_model()` → shows active model
 - Calls `list_sample_files()` → loads benchmark samples

handleModelChange(modelId) (async)

- **Purpose:** Switches to a different Whisper model
- **Called:** When user selects model from dropdown
- **Actions:**
 - Shows loading overlay
 - Calls `switch_model(modelId)` on backend
 - Updates tray icon to "processing"
 - Refreshes backend info (GPU might change)

- Updates UI with success/error message

formatSize(sizeMb)

- **Purpose:** Formats file size (MB → GB conversion)
- **Returns:** "75 MB" or "1.5 GB"
- **Example:** `formatSize(1536)` → "1.5 GB"

setTrayState(newState) (async)

- **Purpose:** Updates tray icon color
- **States:** "ready" (green), "recording" (red), "processing" (yellow)
- **Called:** Before/after recording, model switching

Event Listeners (useEffect hooks)

Hotkey Start Listener

- **Event:** `hotkey-start-recording`
- **Trigger:** User presses Ctrl+Win
- **Actions:**
 - Debounces duplicate events (500ms window)
 - Prevents starting if already recording
 - Calls `start_recording()`
 - Handles pending stop requests

Hotkey Stop Listener

- **Event:** `hotkey-stop-recording`
- **Trigger:** User releases Ctrl+Win
- **Actions:**
 - Queues stop if still starting
 - Calls `stop_recording()`
 - Handles race conditions gracefully

src/App.css (9110 bytes)

Purpose: Styling for the entire application

Key CSS Classes

.container

- Main app container with glassmorphism effect
- Dark background with blur
- Centered layout

.status-bar

- Displays GPU backend and current model
- Color-coded indicators

.model-select

- Dropdown for model selection
- Gradient border animation on hover

.btn-start, .btn-stop, .btn-benchmark

- Recording control buttons
- Animated hover effects
- Disabled states

.loading-overlay

- Full-screen loading indicator
- Spinning animation
- Blur background

🦀 Backend Files (Rust)

src-tauri/src/lib.rs (913 lines)

Purpose: Main orchestrator - handles all Tauri commands, threading, and audio processing

Structs

SendStream(cpal::Stream) (lines 41-44)

- **Purpose:** Wrapper to make audio stream thread-safe
- **Why:** cpal::Stream isn't Send/Sync by default
- **Safety:** We only drop it, never access across threads

AudioState (lines 45-51)

```
pub struct AudioState {
    recording_handle: Arc<Mutex<Option<RecordingHandle>>>,
    whisper: Arc<Mutex<WhisperManager>>,
}
```

- **Purpose:** Shared state across Tauri commands
- **recording_handle:** Currently active recording (if any)
- **whisper:** Whisper AI manager (shared across threads)

RecordingHandle (lines 53-57)

```
struct RecordingHandle {
    stream: SendStream,           // Audio stream (keeps mic active)
    file_tx: Sender,              // Channel to file writer thread
    whisper_tx: Sender,           // Channel to Whisper AI thread
}
```

- **Purpose:** Holds resources for active recording
- **Cleanup:** Dropping this stops recording automatically

Tauri Commands (Functions callable from JavaScript)

`greet(name: &str) → String` (lines 59-62)

- **Purpose:** Demo function from Tauri template
- **Not used:** Can be removed

`get_backend_info(state) → Result<String, String>` (lines 64-68)

- **Purpose:** Returns GPU backend being used
- **Returns:** "Backend: CUDA" or "Backend: CPU"
- **Called:** On app start and after model switch

`list_models() → Result<Vec<ModelInfo>, String>` (lines 70-74)

- **Purpose:** Lists all available Whisper models
- **Returns:** Array of model metadata (id, name, size)
- **Called:** On app start to populate dropdown

`get_current_model(state) → Result<Option<String>, String>` (lines 76-81)

- **Purpose:** Gets currently loaded model name
- **Returns:** "tiny.en-q5_1" or None
- **Called:** On app start

`switch_model(state, model_id) → Result<String, String>` (lines 83-97)

- **Purpose:** Switches to a different Whisper model
- **Checks:** Can't switch while recording
- **Actions:**
 - Clears context (fresh start)
 - Calls `whisper.initialize(model_id)`
 - Returns backend info
- **Called:** When user selects new model

`set_tray_state(app, state, new_state) → Result<(), String>` (lines 99-120)

- **Purpose:** Updates tray icon color
- **States:**
 - "ready" → green circle
 - "recording" → red circle
 - "processing" → yellow circle
- **Called:** Throughout recording lifecycle

`update_tray_icon(app, state) → Result<(), String>` (lines 122-147)

- **Purpose:** Helper to actually change the tray icon

- **Uses:** Built-in tray icons (emoji-red_circle.ico, etc.)

`list_sample_files() → Result<Vec<SampleFile>, String>` (lines 155-209)

- **Purpose:** Lists WAV files in samples directory
- **Returns:** File names and paths for benchmarking
- **Called:** On app start

`benchmark_test(state, file_path) → Result<String, String>` (lines 211-390)

- **Purpose:** Tests transcription performance on sample file
- **Process:**
 1. Load sample audio file
 2. Simulate real-time chunks (6s each)
 3. Run final transcription
 4. Compare performance
- **Returns:** Detailed timing breakdown
- **Called:** When user clicks "Run Benchmark"

Core Recording Functions

`get_recordings_dir() → Result<PathBuf, String>` (lines 392-406)

- **Purpose:** Gets/creates AppData directory for recordings
- **Path:** `C:\Users\YOU\AppData\Local\Taurscribe\temp\`
- **Creates:** Directory if it doesn't exist
- **Called:** By `start_recording()`

`start_recording(state) → Result<String, String>` (lines 408-647)

- **Purpose:** Main recording orchestrator
- **Process:**
 1. Get microphone device
 2. Create WAV file in AppData
 3. Create two channels (file, whisper)
 4. Spawn file writer thread
 5. Spawn Whisper AI thread
 6. Build audio stream with callback
 7. Start recording
 8. Save handle for cleanup
- **Returns:** Success message
- **Called:** When user starts recording

Audio Callback (inside `start_recording`) (lines 290-322)

- **Runs:** Every ~10ms when audio available
- **Actions:**
 1. Receive audio samples from microphone
 2. Convert stereo → mono
 3. Send stereo to file channel

- 4. Send mono to Whisper channel

- **Purpose:** Splits audio into two streams

File Writer Thread (lines 162-172)

- **Runs:** In background, parallel to main thread
- **Process:**
 - Waits for audio on channel
 - Writes samples to WAV file
 - Continues until channel closes
 - Finalizes WAV file on exit
- **Purpose:** Saves all audio to disk

Whisper AI Thread (lines 180-285)

- **Runs:** In background, parallel to main thread
- **Process:**
 - Buffers audio in memory
 - When 6 seconds accumulated:
 - Extract chunk
 - Transcribe with Whisper
 - Print to console
 - Update context
 - Repeat until channel closes
- **Purpose:** Live transcription preview

stop_recording(state) → Result<String, String> (lines 649-722)

- **Purpose:** Stops recording and runs final transcription
- **Process:**
 1. Get recording handle
 2. Drop it (closes channels, stops threads)
 3. Wait for WAV file to finalize
 4. Run `transcribe_file()` on saved audio
 5. Return final transcript
- **Returns:** Complete transcription
- **Called:** When user stops recording

start_hotkey_listener(app_handle) (lines 724-797)

- **Purpose:** Listens for Ctrl+Win global hotkey
- **Process:**
 - Spawns background thread
 - Uses `rdev` to detect key events
 - Tracks Ctrl and Win key states
 - Emits events to frontend:
 - `hotkey-start-recording` (both pressed)
 - `hotkey-stop-recording` (either released)
- **Called:** Once on app startup

run() (lines 799-912)

- **Purpose:** Main application entry point
 - **Process:**
 1. Initialize Whisper manager
 2. Load default model
 3. Create audio state
 4. Build Tauri app
 5. Register all commands
 6. Setup tray icon
 7. Start hotkey listener
 8. Run event loop
 - **Called:** By `main.rs` on app start
-

src-tauri/src/whisper.rs (735 lines)

Purpose: Whisper AI manager - handles model loading, transcription, and audio preprocessing

Enums**GpuBackend** (lines 14-19)

```
pub enum GpuBackend {
    Cuda,      // NVIDIA GPUs
    Vulkan,    // AMD/Intel/Universal
    Cpu,       // Fallback
}
```

- **Purpose:** Tracks which GPU backend is active
- **Display:** Implements `Display` trait for pretty printing

Structs**ModelInfo** (lines 32-38)

```
pub struct ModelInfo {
    pub id: String,           // "tiny.en-q5_1"
    pub display_name: String, // "Tiny English (Q5_1)"
    pub file_name: String,    // "ggml-tiny.en-q5_1.bin"
    pub size_mb: f32,         // 75.0
}
```

- **Purpose:** Metadata about available models
- **Serializable:** Can be sent to JavaScript

WhisperManager (lines 41-46)

```
pub struct WhisperManager {
    context: Option<WhisperContext>, // Loaded model
    last_transcript: String, // Cumulative context
    backend: GpuBackend, // Active backend
    current_model: Option<String>, // Model ID
}
```

- **Purpose:** Manages Whisper model lifecycle and transcription

Functions

`null_log_callback(_level, _text, _user_data)` (lines 48-51)

- **Purpose:** Silences verbose whisper.cpp logs
- **Why:** Prevents console spam
- **Used:** In `initialize()`

`WhisperManager::new() → Self` (lines 54-62)

- **Purpose:** Creates empty manager (no model loaded)
- **Returns:** Manager with None context
- **Called:** Once on app startup

`WhisperManager::get_models_dir() → Result<PathBuf>` (lines 64-82)

- **Purpose:** Finds the models directory
- **Tries:**
 - `taurscribe-runtime/models` (dev)
 - `../taurscribe-runtime/models` (build)
 - `../../taurscribe-runtime/models` (other)
- **Returns:** Canonical path or error

`WhisperManager::list_available_models() → Result<Vec<ModelInfo>>` (lines 84-130)

- **Purpose:** Scans models directory for .bin files
- **Process:**
 1. Read directory
 2. Filter for `ggml-* .bin` files
 3. Parse model ID from filename
 4. Get file size
 5. Format display name
 6. Sort by size
- **Returns:** Array of model metadata

`WhisperManager::format_model_name(id) → String` (lines 132-174)

- **Purpose:** Converts model ID to human-readable name
- **Examples:**
 - `"tiny.en-q5_1"` → `"Tiny English (Q5_1)"`

- "base-q5_0" → "Base Multilingual (Q5_0)"
- "large-v3-turbo" → "Large V3 Turbo Multilingual"

`WhisperManager::get_current_model() → Option<&String>` (lines 176-179)

- **Purpose:** Returns currently loaded model ID
- **Returns:** Reference to model name or None

`WhisperManager::get_backend() → &GpuBackend` (lines 181-184)

- **Purpose:** Returns active GPU backend
- **Returns:** Reference to backend enum

`WhisperManager::clear_context()` (lines 186-190)

- **Purpose:** Resets cumulative transcript
- **When:** Before starting new recording
- **Effect:** Next chunk has no prior context

`WhisperManager::initialize(model_id) → Result<String>` (lines 192-240)

- **Purpose:** Loads Whisper model with GPU acceleration
- **Process:**
 1. Suppress logs
 2. Find model file
 3. Try GPU (CUDA/Vulkan)
 4. Fallback to CPU if GPU fails
 5. Warm-up pass (1s silence)
 6. Store model and backend
- **Returns:** Backend info string
- **Called:** On app start and model switch

`WhisperManager::try_gpu(model_path) → Result<(Context, Backend)>` (lines 242-265)

- **Purpose:** Attempts to load model with GPU
- **Process:**
 1. Enable GPU in parameters
 2. Load model
 3. Detect which backend (CUDA vs Vulkan)
 4. Return context and backend
- **Returns:** Success or error

`WhisperManager::detect_gpu_backend() → GpuBackend` (lines 267-278)

- **Purpose:** Determines if using CUDA or Vulkan
- **Method:** Checks for `nvidia-smi` command
- **Logic:**
 - `nvidia-smi` exists → CUDA
 - Otherwise → Vulkan

`WhisperManager::is_cuda_available() → bool` (lines 280-287)

- **Purpose:** Checks if NVIDIA GPU present
- **Method:** Runs `nvidia-smi` command
- **Returns:** true if successful

`WhisperManager::try_cpu(model_path) → Result<(Context, Backend)>` (lines 289-305)

- **Purpose:** Loads model with CPU (fallback)
- **Process:** Same as GPU but no acceleration
- **Returns:** Context with CPU backend

`WhisperManager::transcribe_chunk(samples, sample_rate) → Result<String>` (lines 307-417)

- **Purpose:** Transcribes a 6-second audio chunk (real-time)
- **Process:**
 1. Resample to 16kHz if needed
 2. Create Whisper state
 3. Set parameters (threads, language, context)
 4. Run inference
 5. Extract transcript
 6. Update cumulative context
 7. Log performance
- **Returns:** Transcribed text
- **Called:** By Whisper AI thread every 6s

`WhisperManager::transcribe_file(file_path) → Result<String>` (lines 419-601)

- **Purpose:** Transcribes complete WAV file (final, high-quality)
- **Process:**
 1. **STEP 1:** Load WAV file
 2. **STEP 2:** Convert stereo → mono
 3. **STEP 3:** Resample to 16kHz
 4. **STEP 4:** Create Whisper state
 5. **STEP 5:** Set optimized parameters
 6. **STEP 6:** Run full inference
 7. **STEP 7:** Extract all segments
 8. **STEP 8:** Combine and return
- **Returns:** Complete transcript
- **Logs:** Detailed timing breakdown
- **Called:** By `stop_recording()`

`WhisperManager::transcribe_audio_data(audio_data) → Result<String>` (lines 602-659)

- **Purpose:** Transcribes pre-processed 16kHz mono audio
- **Similar to:** `transcribe_file` but skips preprocessing
- **Used by:** Benchmark tests
- **Returns:** Transcript

`WhisperManager::load_audio(file_path) → Result<Vec<f32>>` (lines 660-733)

- **Purpose:** Loads and preprocesses WAV file

- **Process:**
 1. Open WAV file
 2. Read all samples
 3. Convert stereo → mono
 4. Resample to 16kHz
 - **Returns:** Ready-to-transcribe audio
 - **Called:** By benchmark function
-

src-tauri/src/vad.rs (186 lines)

Purpose: Voice Activity Detection - filters silence from audio

Struct

VADManager (lines 8-10)

```
pub struct VADManager {
    threshold: f32, // Energy threshold for speech detection
}
```

- **Purpose:** Simple energy-based VAD
- **Note:** Placeholder for future Silero VAD integration

Functions

VADManager::new() → Result<Self> (lines 13-35)

- **Purpose:** Creates VAD manager
- **Process:**
 1. Find models directory
 2. Check for `silero_vad.onnx`
 3. Set energy threshold (0.005)
- **Returns:** Manager ready to detect speech

VADManager::get_models_dir() → Result<PathBuf> (lines 37-54)

- **Purpose:** Finds models directory (same logic as WhisperManager)
- **Returns:** Path to models

VADManager::is_speech(audio) → Result<f32> (lines 56-77)

- **Purpose:** Checks if audio chunk contains speech
- **Method:**
 1. Calculate RMS (Root Mean Square) energy
 2. Compare to threshold
 3. Return probability (0.0 = silence, 1.0 = speech)
- **Returns:** Speech probability

- **Called:** Every frame during VAD processing

`VADManager::get_speech_timestamps(audio, padding_ms) → Result<Vec<(f32, f32)>>` (lines 79-184)

- **Purpose:** Extracts speech segments from full audio
 - **Process:**
 1. Process audio in 512-sample frames (~32ms)
 2. Detect speech/silence transitions
 3. Apply padding around speech
 4. Merge overlapping segments
 5. Filter out very short segments (<150ms)
 - **Returns:** Array of (start_time, end_time) tuples
 - **Called:** By benchmark VAD tests
-

src-tauri/src/main.rs (7 lines)

Purpose: Application entry point

```
fn main() {
    tauriscribe_lib::run()
}
```

- **Purpose:** Launches the app
 - **Special:** `windows_subsystem = "windows"` hides console in release builds
-

src-tauri/build.rs (4 lines)

Purpose: Build-time script

```
fn main() {
    tauri_build::build()
}
```

- **Purpose:** Generates Tauri build artifacts
 - **Runs:** Before compilation
-

⚙ Configuration Files

src-tauri/Cargo.toml (50 lines)

Purpose: Rust project configuration and dependencies

Section Breakdown

[package] (lines 1-6)

- **name**: "taurscribe"
- **version**: "0.1.0"
- **edition**: "2021" (Rust edition)

[lib] (lines 10-15)

- **name**: "taurscribe_lib"
- **crate-type**: Library types for Tauri

[dependencies] (lines 20-37)

```

tauri = { version = "2", features = ["tray-icon", "image-png"] }
tauri-plugin-opener = "2"
tauri-plugin-fs = "2"
serde = { version = "1", features = ["derive"] }
cpal = "0.15"                      # Audio I/O
crossbeam-channel = "0.5"           # Thread communication
hound = "3.5"                       # WAV file reading/writing
chrono = "0.4"                      # Timestamps
whisper-rs = { git = "...", features = ["cuda", "vulkan"] }
rubato = "0.14"                     # Audio resampling
dirs = "6.0.0"                      # AppData paths
rdev = "0.5"                        # Global hotkeys

```

[profile.dev] (lines 40-44)

- **opt-level = 0**: Don't optimize your code (fast builds)
- **package."** opt-level = 1**: Lightly optimize dependencies

package.json (27 lines)

Purpose: Node.js/npm project configuration

Scripts

```

"dev": "vite",                  // Start dev server
"build": "tsc && vite build", // Build for production
"preview": "vite preview",      // Preview build
"tauri": "cd src-tauri && cargo check && cd .. && tauri",
"check": "cd src-tauri && cargo check"

```

Dependencies

```
"react": "^19.1.0",           // UI framework
"react-dom": "^19.1.0",
"@tauri-apps/api": "^2",      // Tauri JavaScript API
"@tauri-apps/plugin-opener": "^2"
```

DevDependencies

```
@types/react": "^19.1.8",      // TypeScript types
@vitejs/plugin-react": "^4.6.0", // Vite React plugin
typescript": "~5.8.3",
vite": "^7.0.4",                // Build tool
@tauri-apps/cli": "^2"         // Tauri CLI
```

📐 Module Architecture

This section documents Taurscribe's modular architecture introduced in January 2026.

📁 File Organization Summary

📐 New Modular Architecture (Refactored January 2026)

Taurscribe follows a **feature-based module structure** for maximum maintainability:

```
Taurscribe/
  └── 🎨 Frontend
    ├── src/App.tsx          # Main UI logic (585 lines)
    ├── src/App.css          # Styling
    ├── src/main.tsx         # React entry point
    └── index.html           # HTML shell

  └── 🦸 Backend (Rust) - REFACTORED STRUCTURE
    └── src-tauri/
      └── src/
        └── 🎯 Core
          └── lib.rs            # App entry point (115 lines) ⚡ 90%
reduction!
          ├── main.rs           # Binary entry (6 lines)
          ├── types.rs          # Shared types & enums (30 lines)
          ├── state.rs          # Global AudioState (54 lines)
          └── utils.rs          # Helper functions (46 lines)

          └── 🎧 Audio Processing
            ├── audio.rs         # Audio primitives (16 lines)
            ├── whisper.rs       # Whisper AI manager (780 lines)
            ├── parakeet.rs      # Parakeet AI manager (578 lines)
            └── vad.rs            # Voice Activity Detection (214 lines)
```

```

└── Commands (Tauri API)
    ├── commands/
    │   ├── mod.rs          # Module exports (11 lines)
    │   ├── misc.rs         # Test commands (5 lines)
    │   ├── models.rs        # Model management (60 lines)
    │   ├── settings.rs     # Engine & backend config (55 lines)
    │   ├── recording.rs    # Start/stop recording (344 lines)
    │   └── transcription.rs # Benchmarks & samples (223 lines)

    └── System Tray
        └── tray/
            ├── mod.rs          # Module exports (2 lines)
            └── icons.rs         # Icon management & menu (92 lines)

    └── Global Hotkeys
        └── hotkeys/
            ├── mod.rs          # Module exports (2 lines)
            └── listener.rs      # Ctrl+Win listener (74 lines)

    └── build.rs          # Build script (4 lines)
    └── Cargo.toml        # Rust dependencies (109 lines)

    └── Configuration
        ├── package.json      # Node.js config (29 lines)
        ├── vite.config.ts    # Vite config
        ├── tsconfig.json     # TypeScript config
        └── .gitignore

    └── Runtime Assets
        └── taurscribe-runtime/
            ├── models/          # AI model files (.bin, .onnx)
            └── samples/          # Test audio files (.wav)

    └── Documentation
        ├── ARCHITECTURE.md    # This file!
        └── README.md

```

Module Organization Guide

Why This Structure?

The codebase was refactored from a **monolithic 1,151-line lib.rs** into a clean, modular architecture that:

- Makes code **easy to find** (organized by feature)
- Enables **parallel development** (multiple people can work on different modules)
- Simplifies **testing** (each module is self-contained)
- Reduces **merge conflicts** (changes are isolated)
- Follows **Rust best practices** (single responsibility principle)

Module Breakdown

1 Core Modules (Foundation)

File	Lines	Purpose
lib.rs	115	App entry point - Initializes engines, sets up Tauri, registers commands
types.rs	30	Shared types - AppState, ASREngine, TranscriptionChunk, SampleFile
state.rs	54	Global state - AudioState struct that holds all shared data
utils.rs	46	Helper functions - clean_transcript(), get_recordings_dir()
audio.rs	16	Audio primitives - SendStream, RecordingHandle

Key Concept: These modules provide the foundation that all other modules depend on.

2 AI Engines (Audio Processing)

File	Lines	Purpose
whisper.rs	780	Whisper AI - Model loading, CUDA/Vulkan, chunk transcription, context management
parakeet.rs	578	Parakeet AI - NVIDIA Nemotron, CTC/TDT variants, streaming transcription
vad.rs	214	Voice Activity Detection - Energy-based & Silero model speech detection

Key Concept: These modules are unchanged from the original structure - they were already well-organized!

3 Commands Module (Tauri API Handlers)

The commands/ directory contains all Tauri commands organized by domain:

File	Lines	Handles
misc.rs	5	Test commands (greet)
models.rs	60	Model management: list_models, switch_model, list_parakeet_models, init_parakeet
settings.rs	55	Engine settings: get_backend_info, set_active_engine, set_tray_state
recording.rs	344	Recording lifecycle: start_recording, stop_recording (includes threading logic)
transcription.rs	223	Testing: benchmark_test, list_sample_files

Why Split by Domain?

- Easy to find: "Where's the recording logic?" → recording.rs
- Clear ownership: Each file has a specific responsibility
- Better imports: Only import what you need

Example: Adding a new model-related command?

```
// Go to: src-tauri/src/commands/models.rs
#[tauri::command]
pub fn my_new_model_command(state: State<AudioState>) -> Result<String, String> {
    // Implementation here
}

// Then add to lib.rs invoke_handler:
commands::my_new_model_command,
```

4 Tray Module (System Tray)

File	Lines	Purpose
tray/icons.rs	92	Icon macros, update_tray_icon(), setup_tray() - manages tray icon state

Key Functions:

- `setup_tray()` - Creates tray icon with menu on app startup
- `update_tray_icon()` - Changes icon color based on app state (green/red/yellow)

5 Hotkeys Module (Global Keyboard)

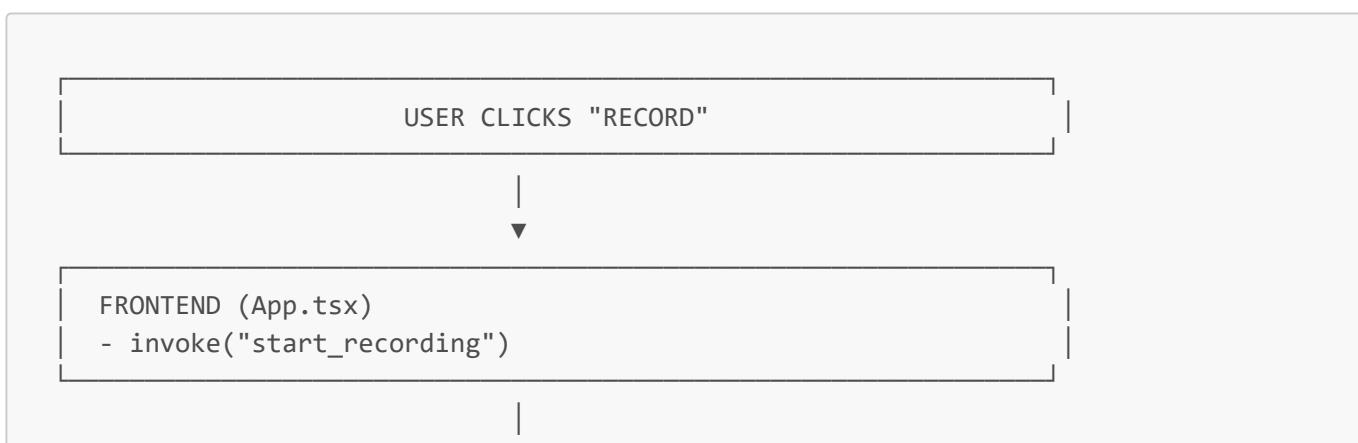
File	Lines	Purpose
hotkeys/listener.rs	74	start_hotkey_listener() - Monitors Ctrl+Win keypresses system-wide

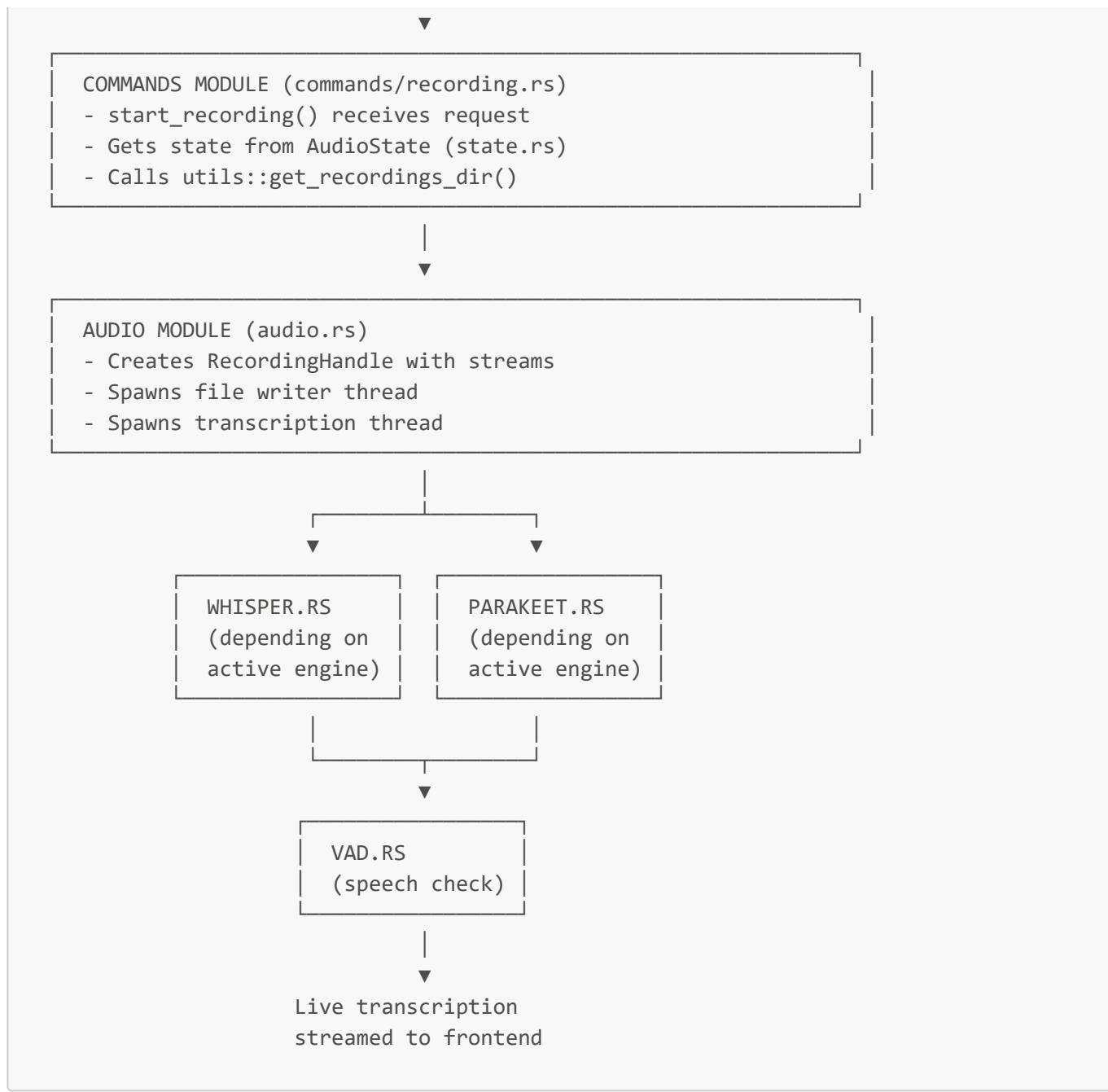
How it works:

1. Spawns background thread
2. Uses `rdev` to listen to all keyboard events
3. Emits Tauri events when Ctrl+Win is pressed/released
4. Frontend listens for these events and triggers recording

6 Data Flow Through Modules

Here's how a recording session flows through the new architecture:





🎓 Module Design Philosophy

Feature-First Organization

Instead of organizing by technical layer (controllers, services, models), we organize by **feature**:

✗ **Bad** (Layer-first):

```

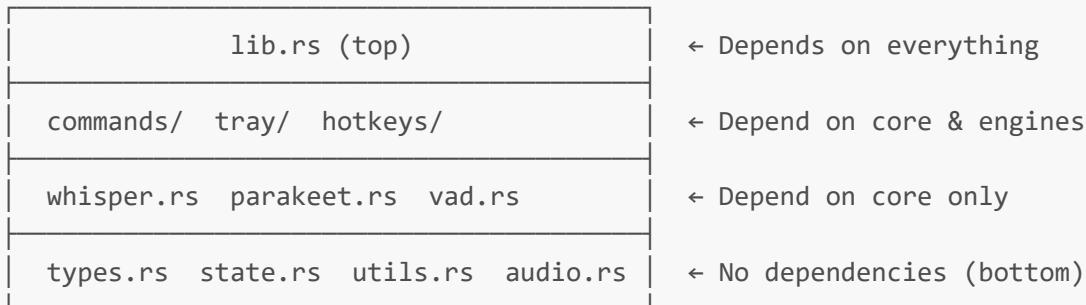
src/
  └── controllers/
  └── services/
  └── models/
  
```

☑ **Good** (Feature-first):

```
src/
└── commands/      # All user-facing commands
└── audio/         # All audio processing
└── tray/          # All tray functionality
└── hotkeys/       # All hotkey functionality
```

Why? When you need to modify "recording", all recording-related code is in one place!

Dependency Rules



Rule: Lower modules never depend on higher modules. This prevents circular dependencies!

File Size Guidelines

Size	Status	Action
< 200 lines	<input checked="" type="checkbox"/> Ideal	Perfect size, easy to understand
200-500 lines	<input checked="" type="checkbox"/> Good	Manageable, consider splitting if it grows
500-800 lines	<input type="checkbox"/> Large	Should be split soon, but acceptable for now
> 800 lines	<input checked="" type="checkbox"/> Too Big	Split immediately! Hard to maintain

Current Status: All modules are \leq 780 lines (whisper.rs is the largest, but it's cohesive).

🔍 Quick Function Lookup

Need to add a feature?

Task	File	Function
Change UI layout	App.tsx	Component JSX
Add new button	App.tsx	Add button in render
Add Tauri command	commands/*.rs → lib.rs	Create in appropriate commands file + add to .invoke_handler()

Task	File	Function
Change model behavior	whisper.rs / parakeet.rs	Modify <code>transcribe_chunk()</code> or <code>transcribe_file()</code>
Adjust recording	commands/recording.rs	<code>start_recording()</code> audio callback
Add model command	commands/models.rs	New function + register in <code>lib.rs</code>
Add engine setting	commands/settings.rs	New function + register in <code>lib.rs</code>
Change VAD threshold	vad.rs	VADManager::new() threshold value
Modify tray behavior	tray/icons.rs	<code>update_tray_icon()</code> or <code>setup_tray()</code>
Change hotkey combo	hotkeys/listener.rs	Modify key detection logic
Add helper function	utils.rs	Create new public function
Add shared type	types.rs	Define struct/enum with derive macros
Add dependency	Cargo.toml	Add to <code>[dependencies]</code>
Change styling	App.css	Modify CSS classes

Debugging a feature?

Issue	Check File	Check Function
Recording not starting	commands/recording.rs	<code>start_recording()</code>
No audio in WAV file	commands/recording.rs	File writer thread (line ~601)
Transcription wrong	whisper.rs / parakeet.rs	<code>transcribe_chunk()</code> or <code>transcribe_file()</code>
UI not updating	App.tsx	State setters, event listeners
Model not loading	whisper.rs / parakeet.rs	<code>initialize()</code> , check GPU detection
Model switching fails	commands/models.rs	<code>switch_model()</code> or <code>init_parakeet()</code>
Backend info wrong	commands/settings.rs	<code>get_backend_info()</code>
Hotkey not working	hotkeys/listener.rs	<code>start_hotkey_listener()</code> callback
Tray icon wrong	commands/settings.rs	<code>set_tray_state()</code> → <code>tray/icons.rs</code>
Tray menu not working	tray/icons.rs	<code>setup_tray()</code> event handlers
Audio format issues	audio.rs	<code>RecordingHandle</code> struct
State not persisting	state.rs	<code>AudioState</code> struct fields

Issue	Check File	Check Function
Transcript cleaning	utils.rs	clean_transcript()

Module Cheat Sheet

Quick reference for finding code:

I want to...	Go to
Add a new command	commands/*.rs (choose appropriate file)
Modify recording behavior	commands/recording.rs
Change AI engine logic	whisper.rs or parakeet.rs
Add a new model	commands/models.rs
Change app settings	commands/settings.rs
Modify tray icon	tray/icons.rs
Change hotkey combo	hotkeys/listener.rs
Add shared types	types.rs
Add utility function	utils.rs
Modify global state	state.rs
Change audio handling	audio.rs
Run benchmarks	commands/transcription.rs

Adding a New Feature: Step-by-Step

Example: Adding a "Pause Recording" feature

1. Add State (`state.rs`):

```
pub is_paused: Mutex<bool>,
```

2. Add Command (`commands/recording.rs`):

```
#[tauri::command]
pub fn pause_recording(state: State<AudioState>) -> Result<(), String> {
    *state.is_paused.lock().unwrap() = true;
    Ok(())
}
```

3. Register Command (`lib.rs`):

```
.invoke_handler(tauri::generate_handler![
    // ... existing commands ...
    commands::pause_recording,
])
```

4. Call from Frontend ([App.tsx](#)):

```
await invoke("pause_recording");
```

Done!

LLM Integration: SmoLLM2 Grammar Correction

This section documents the **complete integration of a Transformer-based Large Language Model (LLM)** into Taurscribe for grammar correction. We'll cover every crate, every function, every issue encountered, and how they were solved.

Purpose

After transcription, the text may contain grammatical errors, especially for real-time streaming. SmoLLM2 (a 135M parameter model from Hugging Face) is integrated to:

1. **Correct grammar errors** in transcribed text
 2. **Run locally** (no API calls, full privacy)
 3. **Use GPU acceleration** (CUDA) when available
 4. **Integrate seamlessly** with the existing Tauri architecture
-

Crates Used

Here are ALL the Rust crates we added and why:

1. candle-core (version 0.9.2)

```
candle-core = { version = "0.9.2", default-features = false }
```

What it does: The core tensor library from Hugging Face. Like PyTorch but for Rust.

Key Types Used:

- [DType::F32](#) - 32-bit floating point numbers for model weights
- [Device::Cpu / Device::new_cuda\(0\)](#) - Where computations run
- [Tensor](#) - Multi-dimensional arrays (like NumPy arrays)

Why `default-features = false`?: By default, candle enables CUDA kernel compilation which requires `nvcc`. This fails outside the x64 Native Tools Command Prompt. Disabling default features means we manually control CUDA support.

2. candle-nn (version 0.9.2)

```
candle-nn = { version = "0.9.2", default-features = false }
```

What it does: Neural network building blocks (layers, activations, etc.)

Key Types Used:

- `VarBuilder` - Loads model weights from safetensors files
 - `VarBuilder::from_mmaped_safetensors()` - Memory-maps the file for efficiency
-

3. candle-transformers (version 0.9.2)

```
candle-transformers = { version = "0.9.2", default-features = false }
```

What it does: Pre-built transformer architectures (Llama, GPT, BERT, etc.)

Key Types Used:

- `Llama` - The actual LLM model class
 - `Config` - Model configuration (hidden_size, num_layers, etc.)
 - `Cache` - KV-cache for efficient token generation
 - `LlamaEosToks::Single(u32)` - End-of-sequence token wrapper
 - `LogitsProcessor` - Sampling strategy for text generation
-

4. tokenizers (version 0.21.0)

```
tokenizers = "0.21.0"
```

What it does: Hugging Face's tokenizer library. Converts text ↔ token IDs.

Key Types Used:

- `Tokenizer` - Loads tokenizer.json and handles encoding/decoding
 - `Tokenizer::encode()` - "Hello world" → [1234, 5678, ...]
 - `Tokenizer::decode()` - [1234, 5678, ...] → "Hello world"
 - `Tokenizer::token_to_id()` - Get ID for special tokens like <|im_end|>
-

5. anyhow (version 1.0)

```
anyhow = "1.0"
```

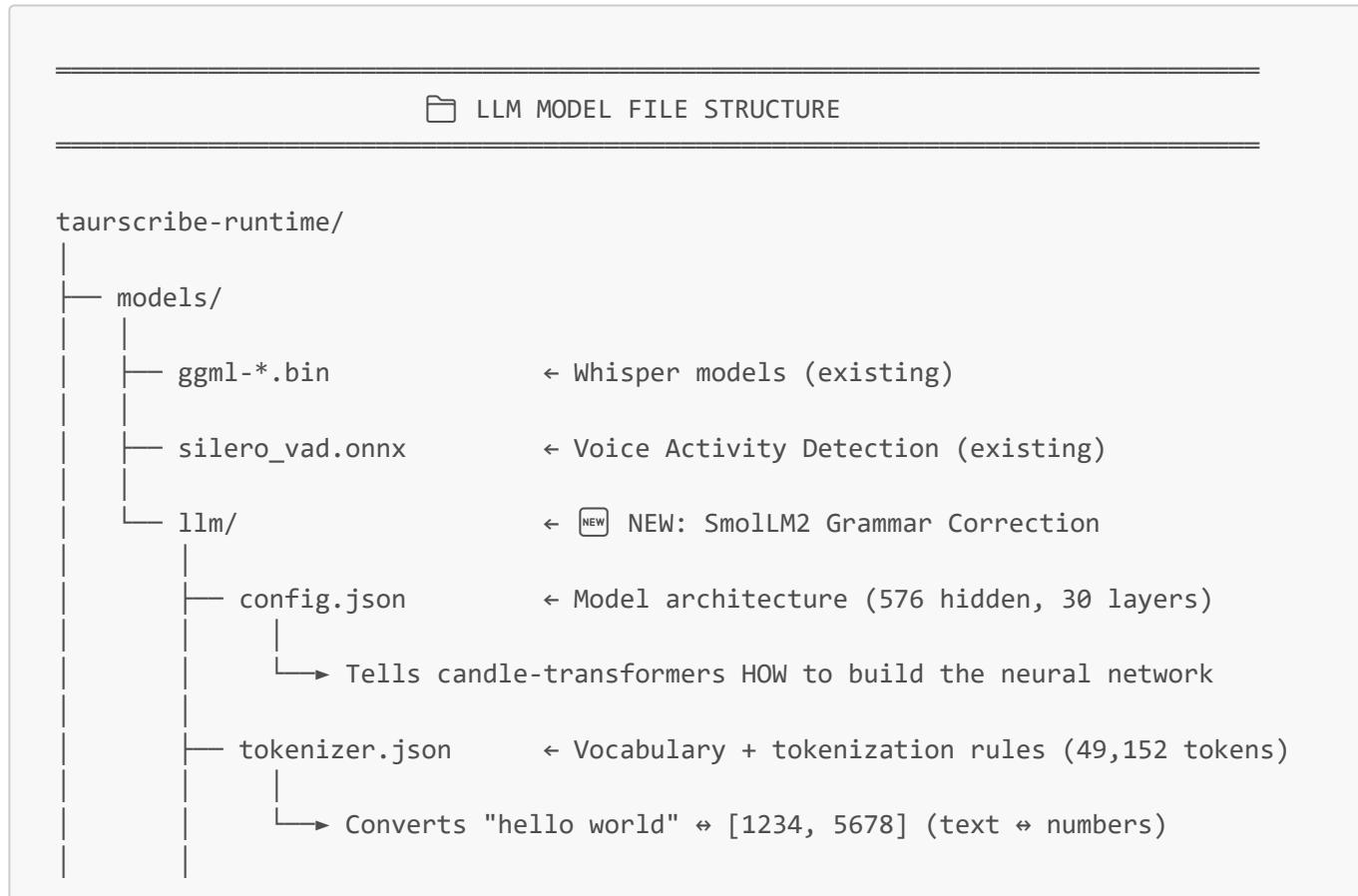
What it does: Simplified error handling with the `Result<T>` type and `anyhow!` macro.

Why?: Candle functions return various error types. `anyhow` lets us convert them all to one unified error type.

📁 Files Created/Modified

File	Purpose
<code>src/llm.rs</code>	NEW - LlmManager struct and all LLM logic
<code>src/lib.rs</code>	Added <code>mod llm;</code> and initialization
<code>src/state.rs</code>	Added <code>llm: Arc<Mutex<LlmManager>></code> to AudioState
<code>src/commands/transcription.rs</code>	Added <code>correct_text</code> Tauri command
<code>Cargo.toml</code>	Added candle, tokenizers, anyhow dependencies
<code>src/App.tsx</code>	Added "Correct Grammar" button and logic
<code>src/App.css</code>	Added <code>.btn-correct</code> styling

📁 Model Files Required & Architecture



```

└── model.safetensors  ← The actual weights (~270 MB, 135M parameters)
    └──→ 135 million numbers that encode "knowledge" of grammar

└── bin/                ← Shared libraries (whisper DLLs, etc.)

```

📄 config.json Explained

config.json (SmolLM2-135M)

```

{
    "hidden_size": 576,           ← Size of each hidden layer (576 neurons)
    "intermediate_size": 1536,   ← Feed-forward expansion (576 → 1536)
    "num_hidden_layers": 30,     ← Number of transformer blocks
    "num_attention_heads": 9,    ← Parallel attention "eyes"
    "num_key_value_heads": 3,    ← Grouped-Query Attention (GQA)
    "vocab_size": 49152,         ← Total tokens in vocabulary
    "max_position_embeddings": 8192, ← Max context length
    "rms_norm_eps": 1e-05,       ← Numerical stability for normalization
    "rope_theta": 100000,        ← Rotary Position Embedding base
    "tie_word_embeddings": true  ← Share input/output embedding weights
}

```



LOADED BY: SmolLM2Config struct (llm.rs)

```

let config_str = std::fs::read_to_string(&config_path)?;
let smol_config: SmolLM2Config = serde_json::from_str(&config_str)?;
let config: Config = smol_config.into(); // Convert to candle format

```

📄 tokenizer.json Explained

tokenizer.json (SmolLM2)

```

{
    "version": "1.0",
    "model": {
        "type": "BPE",           ← Byte-Pair Encoding algorithm
    }
}

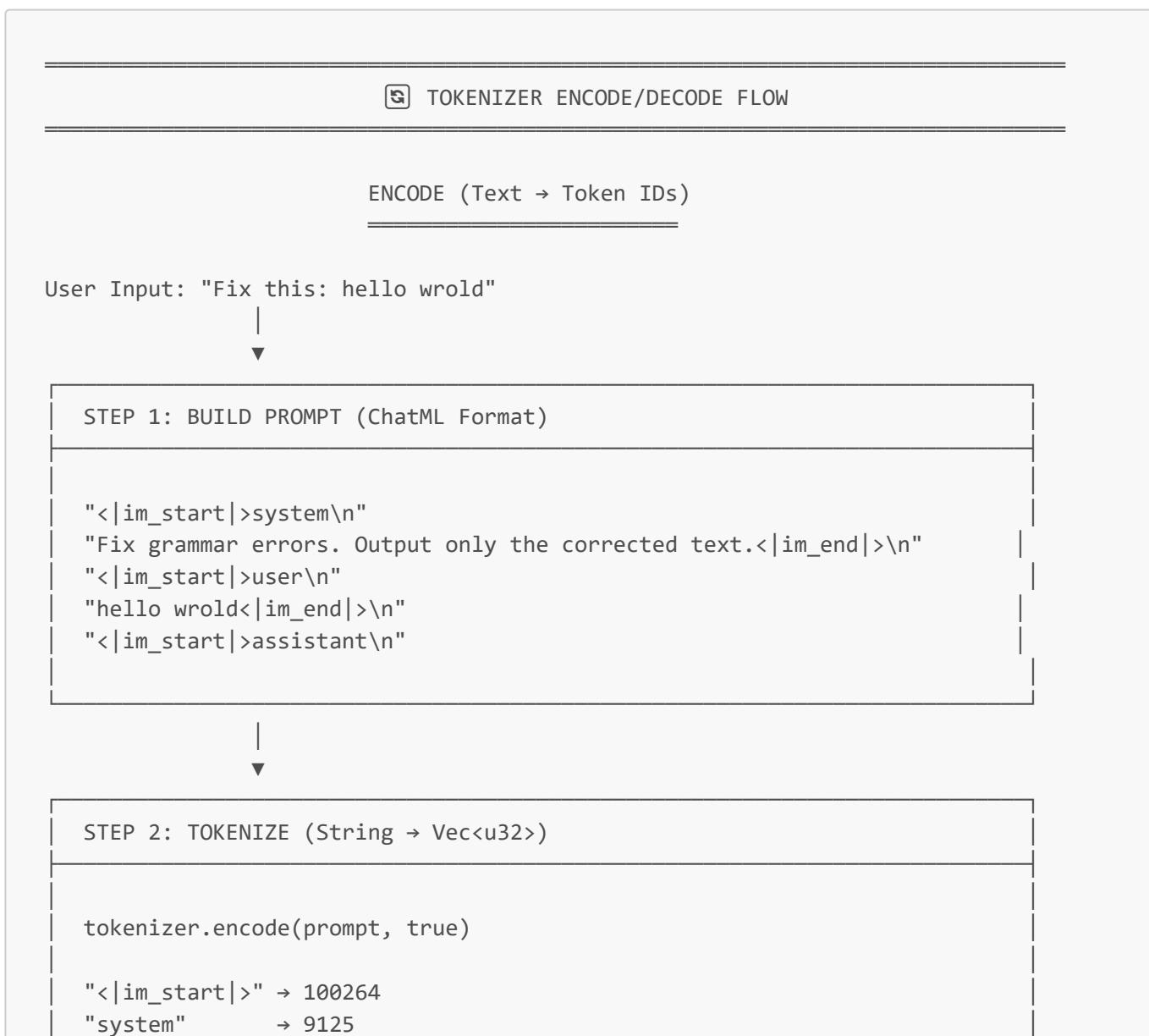
```

```

"vocab": {
    "hello": 1234,           ← Token ID mappings
    "world": 5678,
    "<|im_start|>": 100264, ← Special: Start of message
    "<|im_end|>": 100265,   ← Special: End of message
    ...49,152 more tokens...
},
"merges": [                ← BPE merge rules
    "h e",                 ← Merge 'h' + 'e' → "he"
    "he llo",               ← Merge "he" + "llo" → "hello"
    ...
]
},
"special_tokens_map": {...}
}

```

🔗 Tokenizer Flow Diagram



```
"\n"          → 198
"Fix"        → 22093
" grammar"   → 32548
" errors"    → 6103
"."          → 13
...
"hello"      → 15339
" wr"        → 2949
"old"         → 727           ← "wrold" split into "wr" + "old"
"<|im_end|>" → 100265
```

Result: [100264, 9125, 198, 22093, 32548, 6103, 13, ..., 100265]



STEP 3: FEED TO LLM

```
Token IDs → Tensor::new(&tokens, &device)
→ model.forward(&input, start_pos, cache)
→ Logits (probability distribution over 49,152 tokens)
```

DECODE (Token IDs → Text)

LLM Output: [15496, 995, 13, 100265] (generated tokens)



STEP 4: SAMPLE TOKENS

```
logits_processor.sample(&logits)

Iteration 1: Logits[49152] → Pick token 15496 ("Hello")
Iteration 2: Logits[49152] → Pick token 995 (" world")
Iteration 3: Logits[49152] → Pick token 13 (".")
Iteration 4: Logits[49152] → Pick token 100265 ("<|im_end|>") → STOP!
```

result_tokens = [15496, 995, 13] (excluding <|im_end|>)



STEP 5: DECODE TO TEXT

```
tokenizer.decode(&result_tokens, true)
```

```
15496 → "Hello"
995   → " world"
13    → "."

```

Result: "Hello world."



Final Output: "Hello world." (Grammar corrected!)

📄 model.safetensors Explained

model.safetensors (~270 MB, 135M parameters)

FORMAT: SafeTensors (Hugging Face's safe binary format)

CONTENTS:

model.embed_tokens.weight	[49152, 576]	← Word embeddings 49152 tokens × 576D
model.layers.0.self_attn.q_proj.weight	[576, 576]	← Query matrix
model.layers.0.self_attn.k_proj.weight	[192, 576]	← Key matrix
model.layers.0.self_attn.v_proj.weight	[192, 576]	← Value matrix
model.layers.0.self_attn.o_proj.weight	[576, 576]	← Output matrix
model.layers.0.mlp.gate_proj.weight	[1536, 576]	← FFN gate
model.layers.0.mlp.up_proj.weight	[1536, 576]	← FFN up
model.layers.0.mlp.down_proj.weight	[576, 1536]	← FFN down
model.layers.0.input_layernorm.weight	[576]	← Layer norm
...		
(Repeat for layers 1-29)		
...		
model.layers.29.self_attn.q_proj.weight	[576, 576]	
...		
model.norm.weight	[576]	← Final norm
lm_head.weight	[49152, 576]	← Output layer (tied with embed_tokens)

TOTAL: ~135 million floating-point numbers (f32 = 4 bytes each)

135M × 4 bytes = ~540 MB uncompressed

Actual: ~270 MB (uses float16 in places)



```

LOADED BY: VarBuilder (candle-nn)

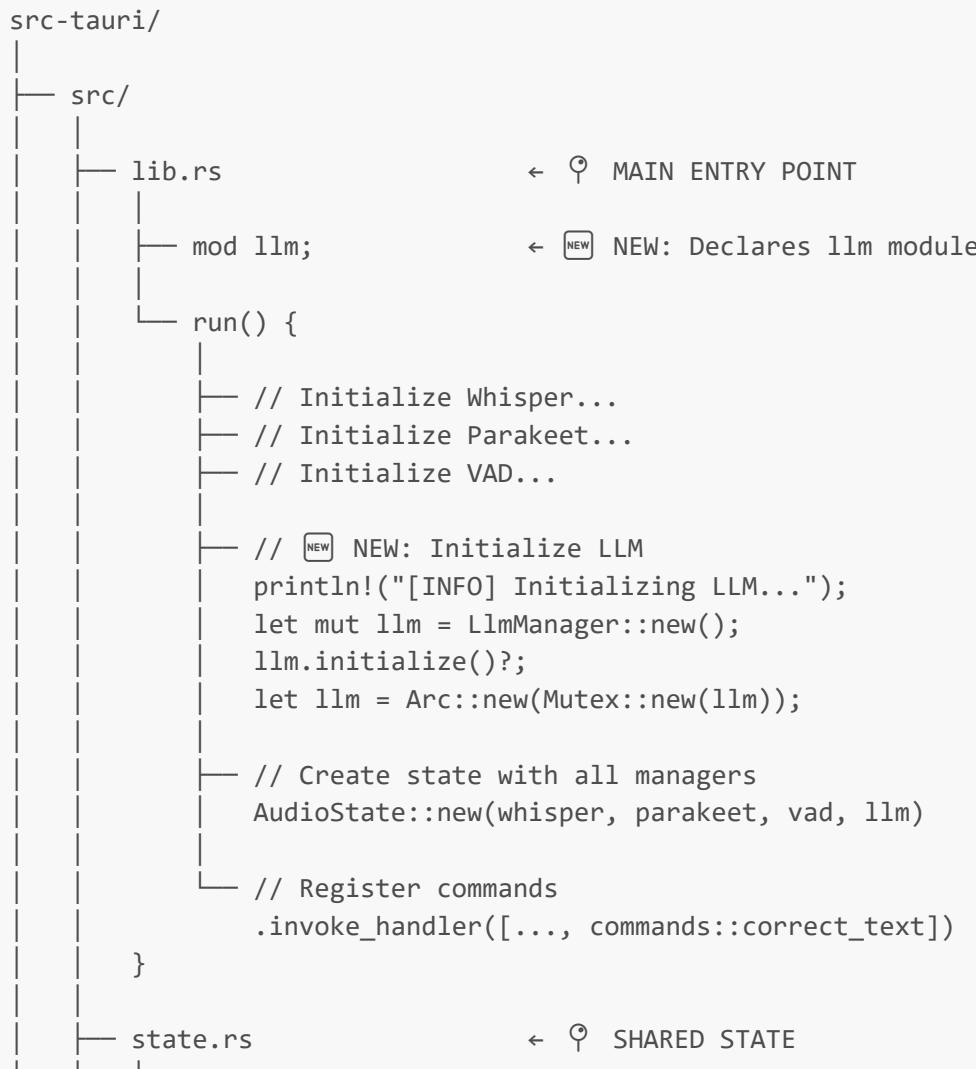
// Memory-map the file (doesn't load all into RAM immediately)
let vb = unsafe {
    VarBuilder::from_mmaped_safetensors(
        &[weights_path],      // Path to model.safetensors
        DType::F32,           // Data type
        &self.device          // CPU or CUDA device
    )?
};

// Build the model architecture and load weights
let model = Llama::load(vb, &config)?;

```

🔗 Codebase Integration Diagram

🔗 LLM INTEGRATION CODEBASE ARCHITECTURE



```

    └── pub struct AudioState {
        ├── whisper: Arc<Mutex<WhisperManager>>,
        ├── parakeet: Arc<Mutex<ParakeetManager>>,
        ├── vad: Arc<Mutex<VadManager>>,
        └── llm: Arc<Mutex<LlmManager>>, ← [NEW] NEW!
    }

    └── llm.rs                                ← [NEW] NEW MODULE (241 lines)
        |
        ├── GpuBackend enum          ← Track CUDA vs CPU
        ├── SmoLLM2Config struct   ← Deserialize config.json
        ├── impl From<SmoLLM2Config> for Config ← Convert to candle
        |
        └── pub struct LlmManager {
            |
            ├── model: Option<Llama>           ← Neural network
            ├── tokenizer: Option<Tokenizer> ← Text ↔ tokens
            ├── cache: Option<Cache>           ← KV-cache
            ├── device: Device                ← CPU or CUDA
            |
            ├── fn get_models_dir()          ← Find llm/ folder
            ├── fn initialize()            ← Load model
            └── fn generate_correction()    ← Run inference
        }

    └── commands/
        └── transcription.rs
            |
            └── #[tauri::command]
                pub fn correct_text(           ← [NEW] NEW COMMAND
                    state: State<AudioState>,
                    text: String
                ) -> Result<String, String> {
                    let mut llm = state.llm.lock()?;
                    llm.generate_correction(&text)
                }
            }

    └── Cargo.toml                            ← ⚙ DEPENDENCIES
        |
        └── [dependencies]
            ├── candle-core = { version = "0.9.2", default-features = false }
            ├── candle-nn = { version = "0.9.2", default-features = false }
            ├── candle-transformers = { version = "0.9.2", default-features = false }
            ├── tokenizers = "0.21.0"
            └── anyhow = "1.0"

    .cargo/config.toml                      ← ⚙ BUILD CONFIG
        |
        └── [env]
            ├── CFLAGS = "/MD /EHsc"      ← [NEW] Fix CRT mismatch
            └── CXXFLAGS = "/MD /EHsc"

```

FRONTEND INTEGRATION

```

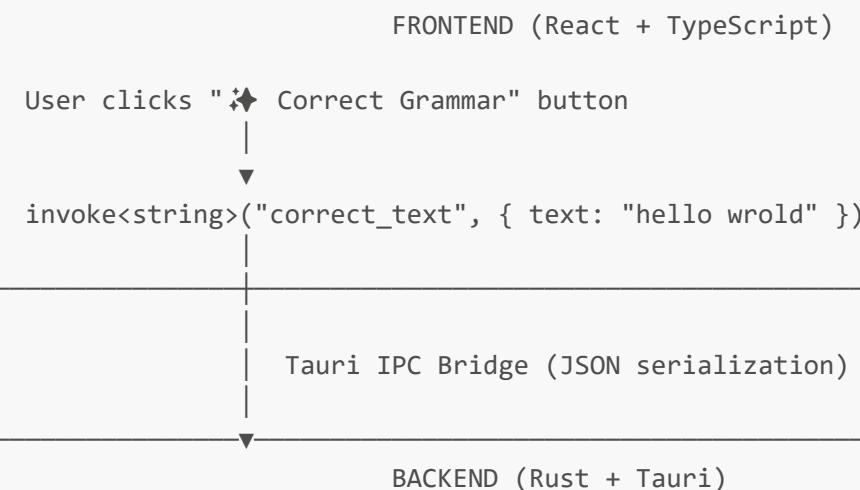
src/
  └── App.tsx           ← ⚡ MAIN REACT COMPONENT
    ├── const [isCorrecting, setIsCorrecting] = useState(false); ← [NEW] NEW
    ├── const handleGrammarCorrection = async () => { ← [NEW] NEW
    │   setIsCorrecting(true);
    │   const corrected = await invoke("correct_text", { text: content });
    │   setContent(corrected);
    │   setIsCorrecting(false);
    │};
    └── <button className="btn-correct" onClick={handleGrammarCorrection}>
        ⚡ Correct Grammar
      </button>

  └── App.css           ← ⚡ STYLES
    └── .btn-correct {      ← [NEW] NEW
      background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
      color: white;
      padding: 8px 16px;
      border-radius: 8px;
      transition: transform 0.2s;
    }

```

⚠ Data Flow: Frontend → Backend → LLM → Frontend

⚠ COMPLETE DATA FLOW DIAGRAM



```
#[tauri::command]
fn correct_text(state: State<AudioState>, text: String)
    |
    | 1. Lock the LlmManager mutex
    |
    |>
let mut llm = state.llm.lock()?;
    |
    | 2. Call generate_correction
    |
    |>
llm.generate_correction(&text)
```

LLM MANAGER (llm.rs)

```
fn generate_correction(&mut self, text: &str) -> Result<String>
    |
    | 3. Build ChatML prompt
    |
    |>
"<|im_start|>system\nFix grammar...<|im_end|>\n..."
```

4. Tokenize

```
tokenizer.encode(prompt, true) -> [100264, 9125, 198, ...]
```

CANDLE (Neural Network)

```
for i in 0..100 {
    |
    | 5. Create tensor from tokens
    |
    |>
    let input = Tensor::new(&tokens[start_pos..], &device)?;
    |
    | 6. Run forward pass (GPU computation!)
    |
    |>
    let logits = model.forward(&input, start_pos, cache)?;
```

INSIDE THE NEURAL NETWORK

```
Tokens → Embeddings (576-dim vectors)
    |
    |>
Layer 1: Attention → MLP
    |
    |>
Layer 2: Attention → MLP
    |
    |>
... (28 more layers)
```



📘 What is a Tokenizer? (Beginner's Guide)

Before the AI can understand text, it needs to convert human words into numbers. That's what a **tokenizer** does!

 TOKENIZER: TEXT ↔ NUMBERS TRANSLATOR

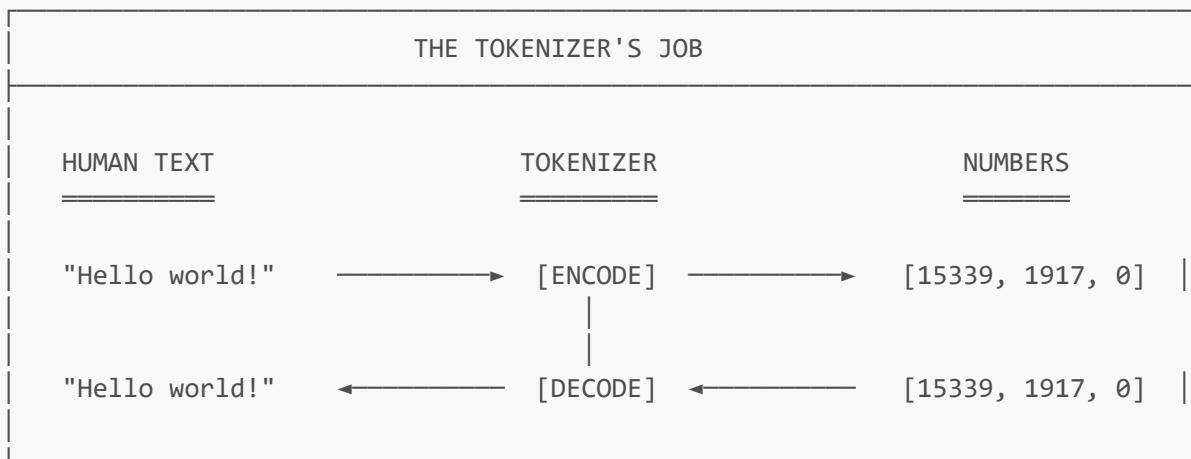
THE PROBLEM:

Computers don't understand words like "hello" or "grammar".
They only understand numbers like 15339 or 49012.

We need a TRANSLATOR to convert between human language and computer language!

WHAT IS A TOKENIZER?

A tokenizer is like a DICTIONARY + TRANSLATOR combined:



THE VOCABULARY (Dictionary)

The tokenizer has a vocabulary of ~49,152 "tokens" (word pieces).

Think of it like a HUGE dictionary where every entry is numbered:

TOKENIZER VOCABULARY (Simplified)

Token ID	Token Text	Type
0	"<unk>"	Unknown token (rare words)
1	"<s>"	Start of sequence
2	"</s>"	End of sequence
...
100264	"< im_start >"	Chat message start (special)
100265	"< im_end >"	Chat message end (special)
...
15339	"Hello"	Common word
1917	" world"	Word with leading space

13	". "	Punctuation
262	" the"	Very common (low ID = frequent)
287	" a"	Very common
...
39421	"Taurscribe"	Rare word (high ID = uncommon)

TOTAL: 49,152 possible tokens

☒ SUBWORD TOKENIZATION (BPE)

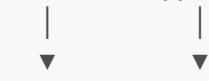
Modern tokenizers don't store every word. They use BPE (Byte-Pair Encoding):

HOW BPE WORKS: "unhappiness" Example

Word: "unhappiness"

Step 1: Break into known pieces

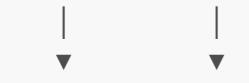
"un" + "happiness"



[982] Is "happiness" known?

NO! Break further:

"happ" + "iness"



[29453] [1619]

Final tokens: [982, 29453, 1619]

"un" + "happ" + "iness"

BENEFIT: Can handle ANY word, even made-up ones like "Taurscribe"!

☒ ENCODE: Text → Token IDs (Step-by-Step)

Example: "Fix grammar: hello wrold"

TOKENIZATION PROCESS

INPUT: "Fix grammar: hello wrold"

Step 1: Add special tokens (ChatML format)

<|im_start|>system

```
| Fix grammar errors and return only the corrected text.<|im_end|>
<|im_start|>user
hello wrold<|im_end|>
<|im_start|>assistant
```

Step 2: Look up each piece in vocabulary

Text Piece	Vocabulary Lookup	Token ID
"< im_start >"	Found!	100264
"system"	Found!	9125
"\n"	Found!	198
"Fix"	Found!	22093
" grammar"	Found!	38428
" errors"	Found!	8563
" and"	Found!	323
" return"	Found!	471
...
"hello"	Found!	15339
" wr"	Found (subword!)	9923
"old"	Found (subword!)	820
"< im_end >"	Found!	100265

OUTPUT: [100264, 9125, 198, 22093, 38428, ..., 15339, 9923, 820, 100265]

⌚ DECODE: Token IDs → Text

The reverse process:

DECODING PROCESS

INPUT: [15339, 1917]

Token ID	Lookup in Vocab	Text Piece
15339	vocab[15339]	"Hello"
1917	vocab[1917]	" world"

Concatenate: "Hello" + " world" = "Hello world"

OUTPUT: "Hello world"

🍕 PIZZA ANALOGY FOR TOKENIZATION

Think of tokenization like ordering pizza:

"I want a large pepperoni pizza with extra cheese"

TOKENIZE (like writing an order slip):

```
Order #15339: Large  
Order #28472: Pepperoni  
Order #19283: Pizza  
Order #92837: Extra Cheese
```

The kitchen (AI) only sees ORDER NUMBERS, not words!

But it knows what each number means because of the menu (vocabulary).

DECODE (reading the order slip back):

```
#15339 → "Large"  
#28472 → "Pepperoni"  
...etc → Original sentence!
```

🧠 What is a Transformer? (Beginner's Guide)

Now that we know how text becomes numbers, let's understand the **brain** that processes them: the **Transformer!**

🧠 TRANSFORMER: THE AI BRAIN ARCHITECTURE

🧠 WHAT PROBLEM DOES IT SOLVE?

Given a sequence of words, predict the NEXT word:

Input: "The cat sat on the ____"
Output: "mat" (most likely next word!)

The Transformer was invented in 2017 and revolutionized AI!
SmolLM2 (used in Taurscribe) is a small Transformer with 135 million "neurons".

🏗 THE BASIC STRUCTURE

A Transformer is like a FACTORY with multiple FLOORS (layers):

TRANSFORMER ARCHITECTURE (SmolLM2)

INPUT: Token IDs [15339, 1917, 0] ("Hello world!")



📘 EMBEDDING LAYER

Convert each token ID into a VECTOR (list of 576 numbers)

Token 15339 → [0.23, -0.45, 0.12, ..., 0.89] (576 numbers)

Token 1917 → [0.56, 0.78, -0.34, ..., -0.12] (576 numbers)

Token 0 → [0.11, 0.22, 0.33, ..., 0.44] (576 numbers)

These vectors ENCODE the meaning of each token!



🧠 LAYER 1: ATTENTION + MLP

— ATTENTION: "Which tokens should I pay attention to?"
(e.g., connect "it" to "cat" in "The cat is fluffy. It...")

— MLP: Transform the information (neural network magic)



🧠 LAYER 2: ATTENTION + MLP



... (28 more layers)



🧠 LAYER 30: ATTENTION + MLP (Final layer!)



📘 OUTPUT LAYER (LM Head)

Convert final vectors back to PROBABILITIES for each possible next token (49,152 options!)

Output: [0.001, 0.002, ..., 0.847, ..., 0.0001]
(49,152 probabilities that sum to 1.0)

Token 1917 ("world") has probability 0.847 = Most likely next!



▼
OUTPUT: Next token = 1917 ("world")

⌚ ATTENTION MECHANISM: THE MAGIC INGREDIENT

Attention is the CORE innovation that makes Transformers so powerful!

PROBLEM: How does the model know that "it" refers to "cat" in:
"The cat sat on the mat. It was fluffy."

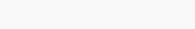
SOLUTION: ATTENTION! The model learns to "look at" relevant tokens.

HOW ATTENTION WORKS (Simplified)

Sentence: "The cat sat on the mat. It was fluffy."

When processing "It", the model asks:
"Which previous words should I pay attention to?"

Attention Weights (learned automatically!):

Word	Attention Score	Visual
"The"	0.02	
"cat"	0.85 ← HIGHEST!	
"sat"	0.03	
"on"	0.01	
"the"	0.02	
"mat"	0.05	
"It"	0.02	

The model learned that "It" most likely refers to "cat"!
This is learned from training data (millions of sentences).

▣ ATTENTION ANALOGY: Reading Comprehension

Imagine you're taking a reading comprehension test:

Passage: "The quick brown fox jumps over the lazy dog.
The dog barks at the fox. The fox runs away."

Question: "What did the dog do?"

Your brain uses ATTENTION:

- You scan the passage
- Focus on "dog" mentions
- Find "The dog barks"
- Answer: "barked at the fox"

THAT'S EXACTLY what the Transformer's attention does,
but with math instead of human intuition!

MULTI-HEAD ATTENTION (SmollM2 has 9 heads)

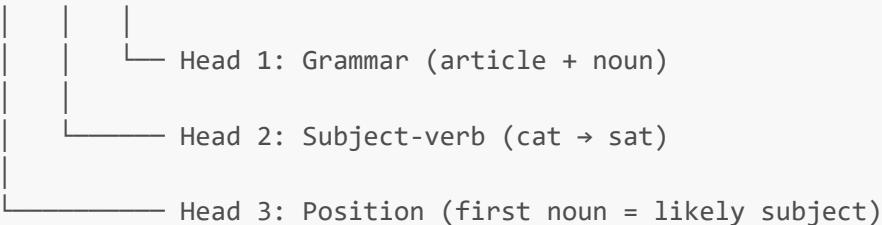
Instead of ONE attention pattern, the model has MULTIPLE "heads":

MULTI-HEAD ATTENTION (9 Heads)

Each head can learn DIFFERENT patterns:

- Head 1: Grammar patterns ("the" usually precedes nouns)
- Head 2: Subject-verb matching (connect "cat" → "is")
- Head 3: Coreference ("it" → "cat")
- Head 4: Position patterns (notice word order)
- Head 5: Semantic similarity (connect related concepts)
- Head 6-9: Other patterns discovered during training

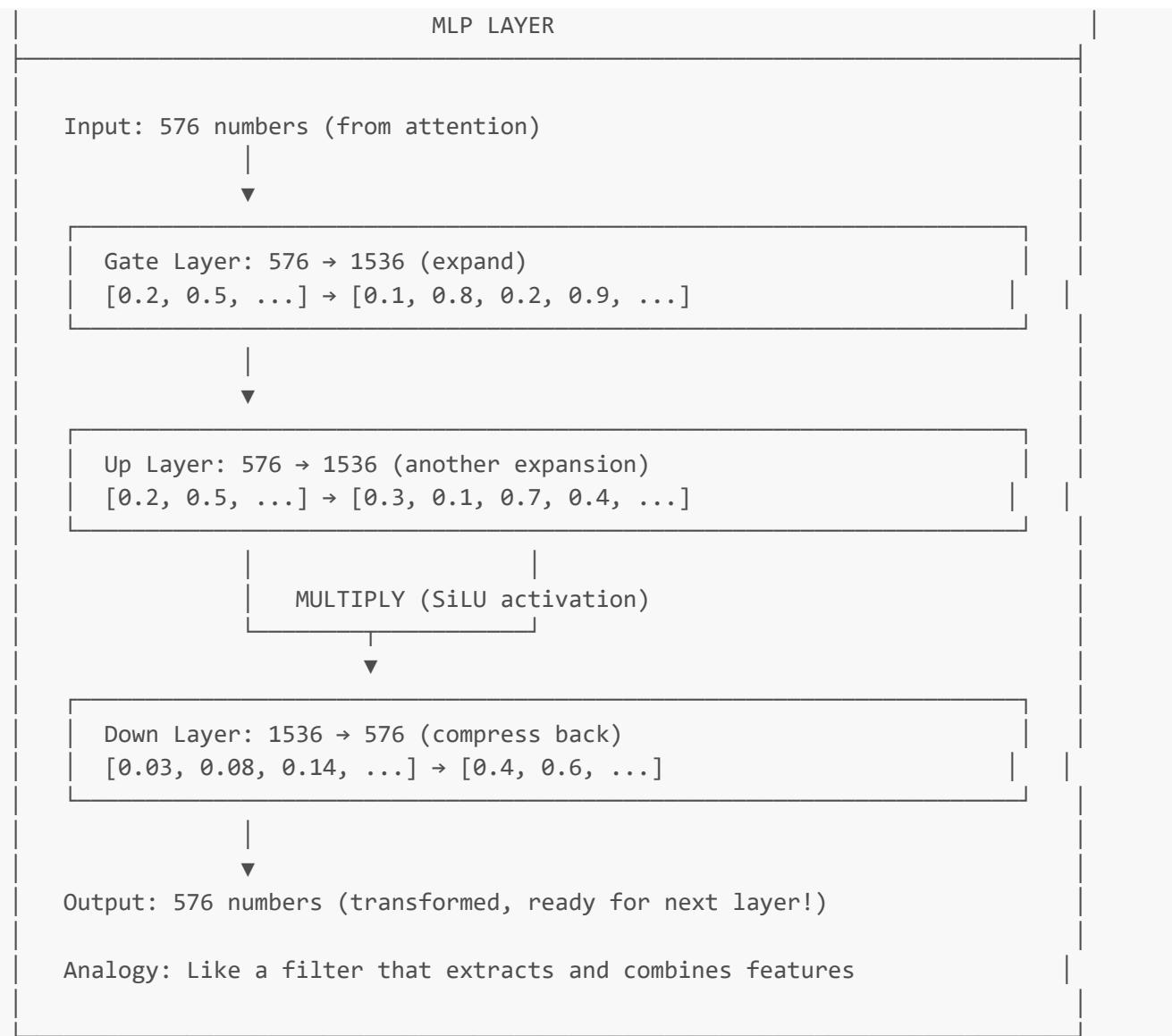
"The cat sat on the mat"



All 9 heads work IN PARALLEL, then combine their findings!

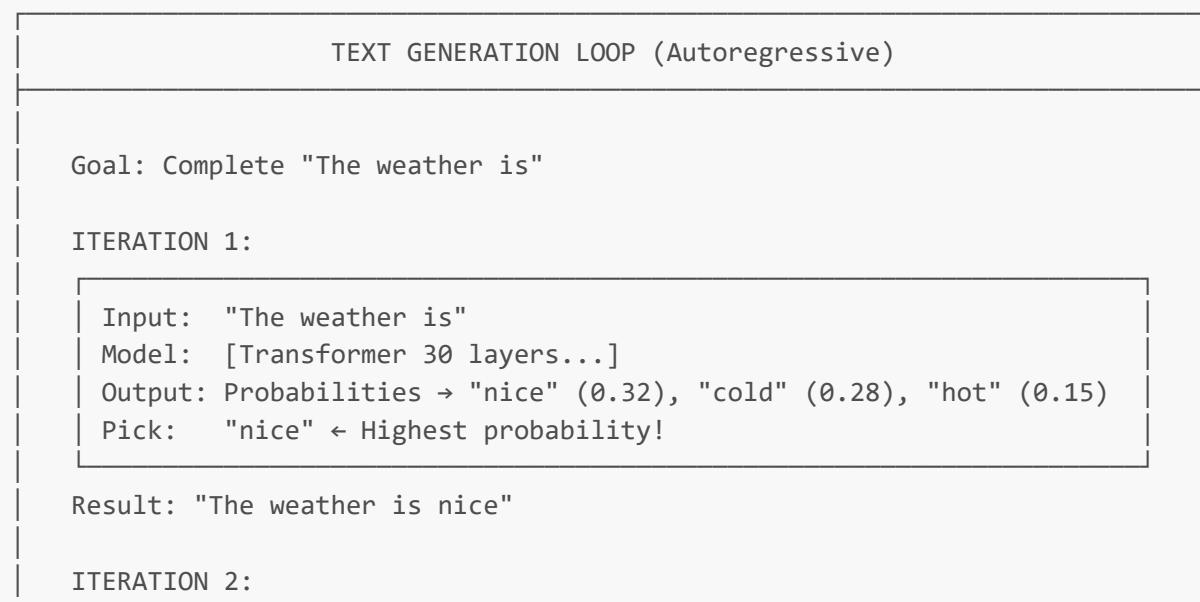
MLP (Multi-Layer Perceptron): The "Thinking" Part

After attention, the MLP transforms the information:



5 THE GENERATION LOOP (Autoregressive)

Transformers generate text ONE TOKEN AT A TIME:



```

Input: "The weather is nice"
Model: [Transformer 30 layers...]
Output: Probabilities → "today" (0.41), "." (0.25), "!" (0.18)
Pick: "today"

```

Result: "The weather is nice today"

ITERATION 3:

```

Input: "The weather is nice today"
Model: [Transformer 30 layers...]
Output: Probabilities → "." (0.67), "!" (0.22), "," (0.08)
Pick: "."

```

Result: "The weather is nice today."

ITERATION 4:

```

Input: "The weather is nice today."
Output: "<|im_end|>" ← END token! STOP generating.

```

FINAL: "The weather is nice today."

SMOLLM2 SPECIFICATIONS (Used in Taurscribe)

SmolLM2-135M-Instruct Stats

ARCHITECTURE:

- Model Type: Llama-style Transformer
- Parameters: 135 Million (135M)
- Layers: 30 transformer blocks
- Hidden Size: 576 dimensions per token
- Attention Heads: 9 heads (parallel attention patterns)
- Key/Value Heads: 3 heads (grouped-query attention for efficiency)
- MLP Size: 1536 (intermediate expansion)
- Vocabulary: 49,152 tokens
- Max Context: 8,192 tokens (~6,000 words)

SIZE COMPARISON:

- GPT-3: 175 BILLION parameters (1,300x larger!)
- GPT-4: ~1.7 TRILLION parameters (12,600x larger!)
- SmolLM2: 135 MILLION parameters (small enough to run locally!)

MEMORY USAGE:

- Model weights: ~270 MB (fp32) or ~135 MB (fp16)
- Runtime RAM: ~500 MB - 1 GB
- GPU VRAM: ~500 MB (fits on almost any GPU!)

PERFORMANCE:

- CPU: ~5-10 tokens/second
- GPU (CUDA): ~50-100 tokens/second
- Grammar correction latency: 1-3 seconds typically

BURGER RESTAURANT ANALOGY

The Transformer is like a fancy burger restaurant:

TRANSFORMER = BURGER RESTAURANT

CUSTOMER ORDER: "hello wrold" (needs grammar correction)

1.  ORDERING SYSTEM (Tokenizer)
"hello wrold" → Order Ticket # [15339, 9923, 820]
2.  INGREDIENT PREP (Embedding Layer)
Each order number → Specific ingredients ready
Token 15339 → [bun, lettuce, tomato, ...]
3.  CHEF STATIONS (30 Transformer Layers)
Each chef adds something:
 - Chef 1: Checks grammar context
 - Chef 2: Understands word relationships
 - Chef 3: Figures out corrections needed
 - ... (30 chefs total!)
4.  ATTENTION (Quality Control)
"Wait, 'wrold' looks wrong. Let me check 'hello' context..."
→ Realizes it should be 'world'!
5.  OUTPUT WINDOW (LM Head)
"Here's your corrected order: 'Hello world.'"
6.  FINAL PRODUCT (Decoded Text)
Token IDs → "Hello world."

🔗 Code Deep Dive: llm.rs

Let's go through EVERY function in detail:

1. GpuBackend Enum

```
#[derive(Debug, Clone, Serialize)]
pub enum GpuBackend {
    Cuda, // NVIDIA GPU acceleration
    Cpu, // Fallback CPU mode
}
```

Why?: Track which compute device we're using. Could be extended to add [Metal](#) (Apple) or [Vulkan](#) in the future.

2. SmoLLM2Config Struct (Hugging Face Format)

```
#[derive(Debug, Clone, Deserialize)]
struct SmoLLM2Config {
    hidden_size: usize, // 576 - Size of hidden layers
    intermediate_size: usize, // 1536 - Size of feed-forward layers
    vocab_size: usize, // 49152 - Number of tokens in vocabulary
    num_hidden_layers: usize, // 30 - Number of transformer blocks
    num_attention_heads: usize, // 9 - Number of attention heads
    num_key_value_heads: Option<usize>, // 3 - For grouped-query attention
    rms_norm_eps: f64, // 1e-05 - Epsilon for RMS normalization
    rope_theta: f32, // 100000 - RoPE position encoding base
    use_flash_attn: bool, // false - Flash attention optimization
    bos_token_id: u32, // 0 - Beginning-of-sequence token ID
    eos_token_id: u32, // 0 - End-of-sequence token ID
    max_position_embeddings: usize, // 8192 - Maximum context length
    tie_word_embeddings: bool, // true - Share input/output embeddings
}
```

Why a separate struct?: Hugging Face config.json has different field names than candle-transformers expects. We deserialize to this struct, then convert to candle's [Config](#).

3. Config Conversion (From Trait)

```
impl From<SmoLLM2Config> for Config {
    fn from(cfg: SmoLLM2Config) -> Self {
        Config {
            hidden_size: cfg.hidden_size,
            intermediate_size: cfg.intermediate_size,
```

```

    vocab_size: cfg.vocab_size,
    num_hidden_layers: cfg.num_hidden_layers,
    num_attention_heads: cfg.num_attention_heads,
    num_key_value_heads:
cfg.num_key_value_heads.unwrap_or(cfg.num_attention_heads),
    rms_norm_eps: cfg.rms_norm_eps,
    rope_theta: cfg.rope_theta,
    use_flash_attn: cfg.use_flash_attn,
    bos_token_id: Some(cfg.bos_token_id),
    eos_token_id: Some(LlamaEosToks::Single(cfg.eos_token_id)), // SPECIAL WRAPPER!
    max_position_embeddings: cfg.max_position_embeddings,
    rope_scaling: None, // No RoPE scaling for SmollM2
    tie_word_embeddings: cfg.tie_word_embeddings,
}
}
}

```

Key Issue Solved: eos_token_id required LlamaEosToks::Single() wrapper because candle supports models with multiple EOS tokens (like Llama 3).

4. LlmManager Struct

```

pub struct LlmManager {
    model: Option<Llama>,           // The actual neural network
    tokenizer: Option<Tokenizer>, // Text ⇄ token converter
    cache: Option<Cache>,           // KV-cache for efficient generation
    device: Device,                  // CPU or CUDA
    backend: GpuBackend,             // Track which device we're using
    model_id: Option<String>,        // "smollm2-135m"
    config: Option<Config>,          // Model configuration
}

```

Design Pattern: All fields are Option<T> because initialization happens after construction. This is a common pattern in Rust for "builder-style" objects.

5. get_models_dir() - Model Discovery

```

fn get_models_dir() -> Result<PathBuf> {
    let possible_paths = [
        "taurscribe-runtime/models/llm",
        "../taurscribe-runtime/models/llm",
        "../../taurscribe-runtime/models/llm",
    ];
    for path in possible_paths {
        if let Ok(canonical) = std::fs::canonicalize(path) {

```

```

        if canonical.is_dir() {
            return Ok(canonical);
        }
    }

    Err(anyhow!("Could not find LLM models directory"))
}

```

Pattern Reused: Same discovery pattern as `WhisperManager::get_models_dir()` and `ParakeetManager::get_models_dir()`. Checks relative paths because the working directory varies between dev and production.

6. initialize() - Model Loading

```

pub fn initialize(&mut self) -> Result<String> {
    let models_dir = Self::get_models_dir()?;
    // Platform-specific CUDA detection
    #[cfg(all(target_os = "windows", target_arch = "x86_64"))]
    {
        if let Ok(device) = Device::new_cuda(0) {
            println!("[LLM] CUDA device available");
            self.device = device;
            self.backend = GpuBackend::Cuda;
        } else {
            println!("[LLM] CUDA not available, using CPU");
            self.device = Device::Cpu;
            self.backend = GpuBackend::Cpu;
        }
    }
    // ... similar for Linux ...

    // Load model files
    let config_str = std::fs::read_to_string(&config_path)?;
    let smol_config: SmolLM2Config = serde_json::from_str(&config_str)?;
    let config: Config = smol_config.into();

    let tokenizer = Tokenizer::from_file(&tokenizer_path)
        .map_err(|e| anyhow!("{}", e))?;

    // Memory-map weights for efficiency (doesn't load entire file into RAM)
    let vb = unsafe {
        VarBuilder::from_mmaped_safetensors(&[weights_path], Dtype::F32,
        &self.device)?
    };

    // Create the model (loads weights into GPU/CPU memory)
    let model = Llama::load(vb, &config)?;
}

```

```
// Create KV-cache for efficient token generation
let cache = Cache::new(true, DType::F32, &config, &self.device)?;

// Store everything
self.model = Some(model);
self.tokenizer = Some(tokenizer);
self.cache = Some(cache);
self.config = Some(config);
self.model_id = Some("smollm2-135m".to_string());

Ok(format!("SmollM2 loaded on {:?}", self.backend))
}
```

Key Concepts Explained:

1. `#[cfg(all(target_os = "windows", target_arch = "x86_64"))]` - Conditional compilation. This code ONLY runs on Windows x64. Different code runs on Linux, macOS, ARM, etc.
2. `Device::new_cuda(0)` - Tries to create a CUDA device (GPU index 0). Returns `Err` if no NVIDIA GPU or drivers available.
3. `VarBuilder::from_mmaped_safetensors()` - Memory-maps the safetensors file. This is marked `unsafe` because it assumes the file won't change while mapped.
4. `Cache::new(true, DType::F32, &config, &self.device)` - Creates KV-cache for transformer. The `true` means "use cache" (vs. recomputing attention every time).

7. generate_correction() - Text Generation

```
pub fn generate_correction(&mut self, text: &str) -> Result<String> {
    // Get references to model components
    let model = self.model.as_ref().ok_or_else(|| anyhow!("Model not
initialized"))?;
    let tokenizer = self.tokenizer.as_ref().ok_or_else(|| anyhow!("Tokenizer not
initialized"))?;
    let cache = self.cache.as_mut().ok_or_else(|| anyhow!("Cache not
initialized"))?;

    // Build the prompt using ChatML format
    let prompt = format!(
        "<|im_start|>system\nFix grammar errors. Output only the corrected text.
<|im_end|>\n<|im_start|>user\n{}<|im_end|>\n<|im_start|>assistant\n",
        text
    );

    // Tokenize the prompt
    let encoding = tokenizer.encode(prompt, true).map_err(|e| anyhow!("{}", e))?;
    let mut tokens: Vec<u32> = encoding.get_ids().to_vec();
```

```

// Create sampler with temperature=0.3 (low = more deterministic)
let mut logits_processor = LogitsProcessor::new(42, Some(0.3), None);
let mut result_tokens: Vec<u32> = Vec::new();

// Get the end token ID
let eos_id = tokenizer.token_to_id("<|im_end|>").unwrap_or(2);

// Autoregressive generation loop
for i in 0..100 { // Max 100 new tokens
    // Context size: full prompt on first iteration, then just last token
    let context_size = if i > 0 { 1 } else { tokens.len() };
    let start_pos = tokens.len().saturating_sub(context_size);

    // Create input tensor
    let input = Tensor::new(&tokens[start_pos..],
    &self.device)?.unsqueeze(0)?;

    // Run forward pass through the model
    let logits = model.forward(&input, start_pos, cache)?;

    // Get logits for the last token position
    let logits = logits.squeeze(0)?;
    let logits = logits.get(logits.dim(0)? - 1)?;

    // Sample next token
    let next_token = logits_processor.sample(&logits)?;
    tokens.push(next_token);
    result_tokens.push(next_token);

    // Stop if we hit the end token
    if next_token == eos_id {
        break;
    }
}

// Decode tokens back to text
let decoded = tokenizer.decode(&result_tokens, true).map_err(|e| anyhow!("{}",
e))?;

Ok(decoded.trim().to_string())
}

```

Autoregressive Generation Explained:

Initial tokens: [<|im_start|>, system, Fix, grammar, ..., assistant, \n]
↓
Model predicts "The"
↓
Tokens now: [<|im_start|>, system, Fix, grammar, ..., assistant, \n, The]
↓
Model predicts

```
"quick"
↓
Tokens now:      [..., \n, The, quick]
                ↓
                Continue until <|im_end|>
```

KV-Cache Optimization:

- On first iteration: Process ALL tokens (expensive)
 - On subsequent iterations: Process ONLY the new token (cheap)
 - The `cache` stores key/value matrices from previous iterations
 - This is why `start_pos` is passed to `forward()`
-

🔗 Integration with Tauri

1. State Management (state.rs)

```
pub struct AudioState {
    // ... existing fields ...
    pub llm: Arc<Mutex<crate::llm::LlmManager>>, // NEW!
}

impl AudioState {
    pub fn new(
        whisper: Arc<Mutex<WhisperManager>>,
        parakeet: Arc<Mutex<ParakeetManager>>,
        vad: Arc<Mutex<VadManager>>,
        llm: Arc<Mutex<LlmManager>>, // NEW!
    ) -> Self {
        // ...
    }
}
```

Why Arc?: The LlmManager needs to be:

- Shared across threads (`Arc` = Atomic Reference Counting)
 - Mutably accessed by multiple commands (`Mutex` = Mutual Exclusion lock)
-

2. Initialization (lib.rs)

```
pub fn run() {
    // Initialize Whisper...
    // Initialize Parakeet...
    // Initialize VAD...

    // NEW: Initialize LLM
    println!("[INFO] Initializing LLM for grammar correction...");
```

```

let mut llm = LlmManager::new();
match llm.initialize() {
    Ok(msg) => println!("[SUCCESS] {}", msg),
    Err(e) => println!("[WARN] LLM not available: {}", e),
}
let llm = Arc::new(Mutex::new(llm));

// Create state with all managers
let state = AudioState::new(whisper, parakeet, vad, llm);

tauri::Builder::default()
    .manage(state) // Register with Tauri
    .invoke_handler(tauri::generate_handler![
        // ... existing commands ...
        commands::correct_text, // NEW!
    ])
    // ...
}

```

3. Tauri Command (commands/transcription.rs)

```

#[tauri::command]
pub fn correct_text(state: State<AudioState>, text: String) -> Result<String, String> {
    let mut llm = state.llm.lock().map_err(|e| e.to_string())?;
    llm.generate_correction(&text).map_err(|e| e.to_string())
}

```

How Tauri Commands Work:

1. Frontend calls `invoke("correct_text", { text: "hello wrold" })`
2. Tauri deserializes arguments
3. Injects `State<AudioState>` automatically
4. Calls this function
5. Serializes return value back to frontend

4. Frontend Usage (App.tsx)

```

const [isCorrecting, setIsCorrecting] = useState(false);

const handleGrammarCorrection = async () => {
    if (!content.trim()) return;
    setIsCorrecting(true);
    try {
        const corrected = await invoke<string>("correct_text", { text: content });
        setContent(corrected);
    }
}

```

```
    } catch (error) {
      console.error("Grammar correction failed:", error);
    } finally {
      setIsCorrecting(false);
    }
};

// JSX
<button
  className="btn-correct"
  onClick={handleGrammarCorrection}
  disabled={isCorrecting || !content.trim()}
>
  {isCorrecting ? "Correcting..." : "⚡ Correct Grammar"}
</button>
```

⚠ Issues Encountered & Solutions

Issue 1: TokenOutputStream Not Found

```
error[E0433]: failed to resolve: could not find `TokenOutputStream` in
`generation`
```

Solution: candle-transformers 0.9.2 uses a different API. Used `tokenizer.decode()` directly instead.

Issue 2: Missing Config Fields

```
error[E0063]: missing fields `bos_token_id`, `eos_token_id`,
`max_position_embeddings`...
```

Solution: The `candle_transformers::models::llama::Config` struct requires more fields than our initial config. Added all required fields to `From<SmolLM2Config>` implementation.

Issue 3: LlamaEosToks Type Mismatch

```
error[E0308]: mismatched types - expected `LlamaEosToks`, found `u32`
```

Solution: Llama 3 models can have multiple EOS tokens. Wrapped single token in `LlamaEosToks::Single(u32)`.

Issue 4: NVCC Compilation Error

```
nvcc error while compiling "src\\affine.cu"
```

Problem: Candle's default features try to compile CUDA kernels, which requires `nvcc` in PATH. **Solution:** Use `default-features = false` for candle crates:

```
candle-core = { version = "0.9.2", default-features = false }
```

Note: CUDA still works! The pre-compiled CUDA support is still available via cudarc.

Issue 5: Windows Linker Error - Runtime Library Mismatch

```
error LNK2038: mismatch detected for 'RuntimeLibrary':  
value 'MT_StaticRelease' doesn't match value 'MD_DynamicRelease'  
  
LINK : fatal error LNK1319: 1 mismatches detected
```

What Happened: The Windows linker refused to build our application because different parts of the code were trying to use different "power sources" for basic operations like memory management.

The Building Analogy: Two Power Systems

Think of your software project as a **construction site** building an office tower (Taurscribe):

1. The City Power Grid (Dynamic Runtime `/MD`)

- Shared electrical system everyone plugs into
- Efficient because everyone shares one source
- Standard for modern Windows applications
- Files: `msvcrt.lib` (import) / `msvcrt.dll` (runtime)

2. Private Diesel Generator (Static Runtime `/MT`)

- Each contractor brings their own generator
- Self-contained but heavy
- Used for standalone tools
- Files: `libcmt.lib` (everything bundled in)

The Conflict:

- **Our Main App (Rust):** "We're using the City Grid (`/MD`)"
- **Whisper Library:** "Great, we're also on the Grid! (`/MD`)"
- **ESAx Library (inside tokenizers):** "Wait, I brought my own Generator! (`/MT`)"

The **Site Foreman (Windows Linker)** said:

"STOP! You can't wire half the building to the city grid and half to a private generator! If someone allocates memory on the grid and tries to free it via the generator circuit, THE BUILDING EXPLODES!
★"

🔍 Why This Matters (Technical Details)

The **C Runtime Library (CRT)** provides basic services every program needs:

- Memory allocation (`malloc`, `free`)
- File I/O (`fopen`, `fclose`)
- String operations (`strlen`, `strcpy`)
- Math functions (`sin`, `cos`, `sqrt`)

There are 4 versions of the CRT on Windows:

Library	Type	Debug?	Flag
<code>libcmt.lib</code>	Static	No	<code>/MT</code>
<code>libcmtd.lib</code>	Static	Yes	<code>/MTd</code>
<code>msvcrt.lib</code>	Dynamic (import)	No	<code>/MD</code>
<code>msvcrt.d.lib</code>	Dynamic (import)	Yes	<code>/MDd</code>

What went wrong:

1. **Rust** defaults to `/MD` (dynamic) when building DLLs
2. **whisper-rs** (C++ dependency) compiled with `/MD`
3. **esaxx-rs** (C++ dependency) compiled with `/MT`

The Danger: If you mix them:

1. esaxx allocates memory using its static runtime (generator)
2. esaxx hands that pointer to your main app
3. Your main app tries to free() it using the dynamic runtime (grid)
4. CRASH! The grid manager says "I didn't create this memory!"

☑ The Solution: Force Everyone to Use the Grid

Instead of trying to make esaxx rebuild from source with different flags (very difficult), we told the C++ compiler:

"When building ANY C++ dependency (including esaxx), always use the Dynamic Runtime."

Added to `.cargo/config.toml`:

```
[target.x86_64-pc-windows-msvc]
rustflags = [
```

```

"-L",
"C:/Users/abdul/OneDrive/Desktop/Taurscribe/taurscribe-runtime/bin",
"-C", "link-arg=/NODEFAULTLIB:LIBCMT", # ← Ignore the static library
]

[env]
CFLAGS = "/MD"      # ← Force C dependencies to use Dynamic Runtime
CXXFLAGS = "/MD"     # ← Force C++ dependencies to use Dynamic Runtime

```

What each flag does:

1. **/NODEFAULTLIB:LIBCMT**:

- Tells the linker: "If you see instructions to use the static library, ignore them."
- It's like cutting the cord to the diesel generator

2. **CFLAGS = "/MD" and CXXFLAGS = "/MD"**:

- Environment variables that get passed to ALL C/C++ builds
- Forces every library to compile with the Dynamic Runtime flag
- Now when cargo rebuilds esaxx, it uses **/MD** instead of **/MT**

Result:

- Main App: Dynamic (**/MD**)
- Whisper: Dynamic (**/MD**)
- Esaxx: **NOW Dynamic (/MD)** ← Fixed!
- All other C++ deps: Dynamic (**/MD**)

Everyone is now plugged into the same power source! The linker is happy. 🎉

🎓 Key Lessons Learned:

1. Windows DLLs should always use Dynamic Runtime (**/MD**)

- This is the standard and avoids these conflicts

2. Mixing **/MT** and **/MD** in a DLL is undefined behavior

- Even if it links, it can crash at runtime

3. Cargo build scripts respect **CFLAGS/CXXFLAGS**

- Setting these in **.cargo/config.toml** affects ALL C/C++ dependencies
- This is how we fixed esaxx without modifying its source code

4. The linker is protecting you!

- **LNK1319** seems annoying, but it prevents memory corruption bugs
- Always take linker errors seriously on Windows

💡 Alternative Approaches We Tried (and why they failed):

Attempt 1: Use `/NODEFAULTLIB:LIBCMT` alone

```
rustflags = ["-C", "link-arg=/NODEFAULTLIB:LIBCMT"]
```

✗ Failed: The linker still detected the metadata mismatch in the `.obj` files before even looking at libraries.

Attempt 2: Force everything to Static (`+crt-static`)

```
rustflags = ["-C", "target-feature=+crt-static"]
```

✗ Failed catastrophically: 90 new linker errors!

- Missing symbols: `__imp_tgammaf`, `__imp_copysignf`, `__imp_modff`, etc.
- Why? Whisper and ONNX Runtime were compiled expecting dynamic CRT
- They couldn't find basic math functions in the static library

Attempt 3: Current solution - Force C/C++ to Dynamic

```
CFLAGS = "/MD"
CXXFLAGS = "/MD"
```

Success: Everyone uses the same runtime from the start.

This forces ALL C/C++ dependencies to use dynamic CRT from compilation, not just at link time.

📊 Performance Characteristics

Metric	Value
Model Size	135M parameters
Weights Size	~270 MB (safetensors)
Loading Time	~2-3 seconds (first load)
Inference Speed (GPU)	~50 tokens/second
Inference Speed (CPU)	~5 tokens/second
Max Context Length	8192 tokens
Typical Correction	50-200ms

🧠 How Transformers Work (Simplified)

INPUT: "hello wrold this is a tset"

TOKENIZER

Converts text to numbers that the model understands

"hello" → 1234, "wrold" → 5678, ...

Result: [1234, 5678, 9012, 3456, 7890, 2345]

EMBEDDING LAYER

Converts each token ID to a vector (576 numbers)

1234 → [0.01, -0.02, 0.03, ..., 0.15]

Result: Matrix of shape [6 tokens × 576 dimensions]

TRANSFORMER LAYERS (×30)

ATTENTION

"Which words are related to each other?"

"wrold" pays attention to "hello" (context)

FEED-FORWARD NETWORK (MLP)

"What transformation should we apply?"

576 → 1536 → 576 (expand then compress)

(Repeat 30 times for deep understanding)

OUTPUT LAYER

Predicts the next token

Result: Probability distribution over 49,152 tokens

$P(\text{"world"}) = 0.85, P(\text{"word"}) = 0.10, \dots$

SAMPLING (LogitsProcessor)

Pick the most likely token (with temperature=0.3)

Selected: "Hello"

Then repeat for next token, and next, until <|im_end|>

```
↓  
OUTPUT: "Hello world, this is a test."
```

💡 Future Improvements

1. **Multiple Models:** Support larger SmolLM2 variants (360M, 1.7B)
2. **Streaming Output:** Show corrections as they generate
3. **Context Awareness:** Use previous transcript for better corrections
4. **Custom Instructions:** Let users customize the correction style
5. **DirectML Support:** Add DirectML backend for AMD GPUs on Windows

🎤 COMPLETE AUDIO PIPELINE: From Button Click to Transcription

This section answers ALL your questions about how audio recording and transcription works in Taurscribe!

⌚ Table of Contents

1. [Where Does the Code Start?](#)
2. [How Does Recording Start?](#)
3. [How is Audio Captured?](#)
4. [How is Audio Sent to Whisper?](#)
5. [How is Audio Fed into Whisper Model?](#)
6. [How is Audio Sent to Parakeet?](#)
7. [How is Audio Fed into Parakeet Model?](#)
8. [How is Text Outputted?](#)
9. [Complete Visual Flow](#)

⌚ Where Does the Code Start?

The journey begins when YOU click the microphone button!

⌚ Location: [src/App.tsx \(Line 168\)](#)

```
const res = await invoke("start_recording");
```

What happens here:

STEP 1: USER INTERACTION

You (User) clicks: [🎤 Start Recording]

```

React Event Handler fires:
├ Sets isRecording = true
├ Updates UI (button turns red)
└ Calls: invoke("start_recording")

What is invoke()?
→ Tauri's "magic bridge" function
→ Sends message from JavaScript → Rust backend
→ Like calling a phone: "Hey backend, start recording!"
```

(Travels through Tauri IPC bridge...)

RUST BACKEND RECEIVES THE CALL

⌚ src-tauri/src/commands/recording.rs (Line 12)

```

#[tauri::command]
pub fn start_recording(...)
```

Analogy: You pressing a doorbell (button click) that sends an electric signal (invoke) through wires (Tauri IPC) to ring a bell (Rust function) inside the house (backend).

🎬 How Does Recording Start?

⌚ **Location:** [src-tauri/src/commands/recording.rs](#) (**Lines 13-272**)

The `start_recording` function is the **control center** that sets up everything. Let's break it down step-by-step:

```

#[tauri::command]
pub fn start_recording(
    app_handle: AppHandle,
    state: State<AudioState>,
) -> Result<String, String>
```

Step-by-Step Breakdown:

STEP 1: Setup Microphone (Lines 17-23)

```
let host = cpal::default_host();
```

```
let device = host.default_input_device()

What's happening:
└ Ask the OS: "Which microphone should I use?"
└ Get the default mic (the one you selected in settings)
└ Get its config (sample rate, channels, etc.)
```

Analogy: Finding the restaurant's supplier
"Which farm delivers fresh vegetables?"

STEP 2: Prepare Output File (Lines 25-30)

```
let filename = format!(
    "recording_{}.wav",
    chrono::Utc::now().timestamp()
);
let path = recordings_dir.join(&filename);
```

Example: "recording_1738368000.wav"

Analogy: Getting a fresh notepad
"I need a new notebook to write in!"

STEP 3: Reset AI Context (Lines 32-41)

```
if active_engine == ASREngine::Whisper {
    state.whisper.lock().unwrap().clear_context();
} else {
    state.parakeet.lock().unwrap().clear_context();
}
```

Why? Start fresh for new recording!
The AI needs to "forget" the previous conversation.

Analogy: Erasing the blackboard
"Clear the board before starting new lesson!"

STEP 4: Create Communication Pipes (Lines 54-55)

```
let (file_tx, file_rx) = unbounded::<Vec<f32>>();  
let (whisper_tx, whisper_rx) = unbounded::<Vec<f32>>();  
  
What are these?  
→ CHANNELS! Think: Garden hoses for data  
→ tx = Transmitter (sends data)  
→ rx = Receiver (receives data)
```

Two separate pipes:
1. file_tx → file_rx (saves audio to disk)
2. whisper_tx → whisper_rx (sends to AI)

Analogy: Setting up conveyor belts
Belt 1: Raw food → Freezer
Belt 2: Raw food → Chef (for cooking)

↓ STEP 5: Spawn File Saver Thread (Lines 62-72)

```
std::thread::spawn(move || {  
    let mut writer = writer;  
    while let Ok(samples) = file_rx.recv() {  
        for sample in samples {  
            writer.write_sample(sample).ok();  
        }  
    }  
    writer.finalize().ok();  
});
```

What's happening:
└ Create a NEW background thread
└ This thread WAITS for audio data on file_rx
└ When data arrives, write it to WAV file
└ Runs in parallel with everything else!

Analogy: Hiring a dedicated recorder
"Your only job: Write down everything on tape!"

↓ STEP 6: Spawn Transcriber Thread (Lines 82-235)

```
std::thread::spawn(move || {  
    let mut buffer = Vec::new();  
    while let Ok(samples) = whisper_rx.recv() {  
        buffer.extend(samples);  
    }  
});
```

```

    // Process in chunks...
    if active_engine == ASREngine::Whisper {
        // Whisper path (we'll detail this later)
    } else {
        // Parakeet path (we'll detail this later)
    }
}
});
```

What's happening:

- └ Create ANOTHER background thread
- └ This thread WAITS for audio data on whisper_rx
- └ Collects audio into a buffer (basket)
- └ When buffer is full enough, send to AI!
- └ Emits results back to frontend

Analogy: Hiring a translator

"Listen to everything and translate in real-time!"



STEP 7: Start Microphone Stream (Lines 240-263)

```

let stream = device.build_input_stream(
    &config,
    move |data: &[f32], _: &_| {
        // THIS CALLBACK RUNS EVERY 10ms!
        file_tx_clone.send(data.to_vec()).ok();
        whisper_tx_clone.send(mono_data).ok();
    },
    ...
)?;

stream.play()?; // 🎵 START RECORDING!
```

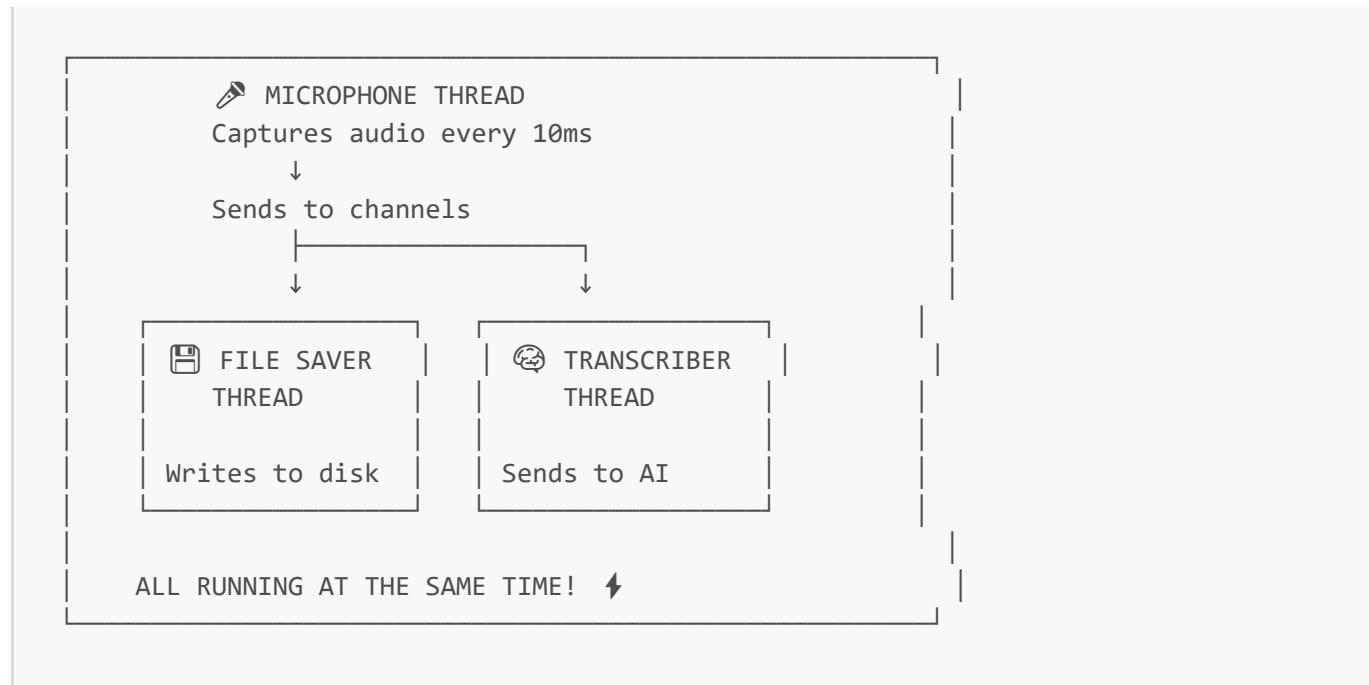
What's happening:

- └ Tell the microphone: "Start capturing!"
- └ EVERY 10 MILLISECONDS, the callback fires
- └ Each time: Get ~480 samples (10ms of audio)
- └ Send copies to BOTH channels (file + transcriber)
- └ This keeps running until we call stream.stop()

Analogy: Opening the water faucet

Water (audio) now flows continuously!

The Result: Three things are now running in parallel!



How is Audio Captured?

⌚ Location: [src-tauri/src/commands/recording.rs](#) (Lines 243-254)

This is the **audio callback** - the heart of the recording system!

```
move |data: &[f32], _: &_| {
    // Send raw audio to file
    file_tx_clone.send(data.to_vec()).ok();

    // Convert stereo → mono
    let mono_data: Vec<f32> = if channels > 1 {
        data.chunks(channels)
            .map(|chunk| chunk.iter().sum::<f32>() / channels as f32)
            .collect()
    } else {
        data.to_vec()
    };

    // Send mono audio to transcriber
    whisper_tx_clone.send(mono_data).ok();
}
```

Breaking It Down:

EVERY 10 MILLISECONDS (100 times per second):

AUDIO CALLBACK FIRES

```
Input: data = [0.001, 0.002, -0.001, 0.003, ...]
      ↑ Array of audio samples (floating point numbers)
      ↑ Each number = microphone membrane position
      ↑ Range: -1.0 (fully in) to +1.0 (fully out)

Typical size: ~480 samples for 10ms at 48kHz
```

STEP 1: Send to File Saver

```
file_tx_clone.send(data.to_vec()).ok();
```

What happens:

1. Clone the audio data (make a copy)
2. Send it through the file_tx channel
3. File saver thread receives it
4. Writes to WAV file

Why clone? We need TWO copies (file + AI)!

STEP 2: Convert Stereo → Mono

Why? AI models expect MONO (1 channel) audio!

Stereo (2 channels):

```
[L1, R1, L2, R2, L3, R3, ...]
↑   ↑   ↑   ↑   ↑   ↑
Left Right ...
```

Conversion Formula: (Left + Right) / 2

```
data.chunks(2) ← Split into pairs: [L1, R1], [L2, R2] ...
    .map(|chunk| {
        (chunk[0] + chunk[1]) / 2.0 ← Average them
    })
```

Result (mono):

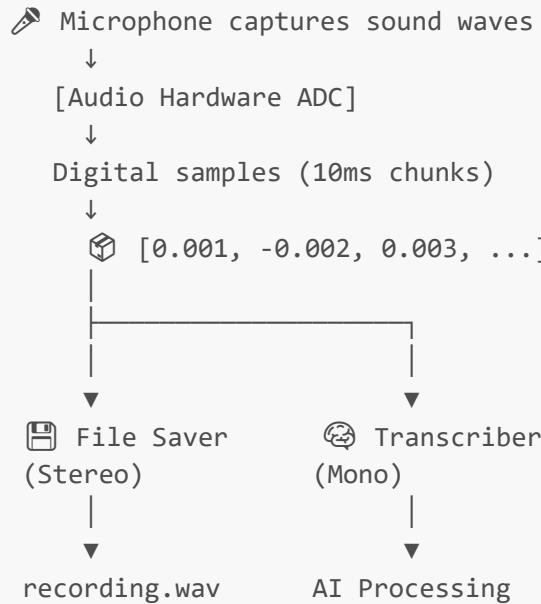
```
[M1, M2, M3, ...]
↑   ↑   ↑
Averaged samples
```

STEP 3: Send to Transcriber

```
whisper_tx_clone.send(mono_data).ok();
```

The mono audio travels through the channel...
Arrives at the Transcriber Thread!

Visual Flow:



Analogy: Security Camera System

- **Camera** = Microphone (captures everything)
- **Video Feed** = Audio stream
- **Recording DVR** = File saver (saves raw footage)
- **AI Analyzer** = Transcriber (processes in real-time)
- **Split Signal** = Sending to both DVR and AI simultaneously

⌚ How is Audio Sent to Whisper?

Whisper has **TWO different paths**: Real-time chunks and Final pass.

Path 1: Real-Time Chunks (During Recording)

⌚ **Location:** [src-tauri/src/commands/recording.rs](#) (**Lines 94-139**)

```
if active_engine == ASREngine::Whisper {
    buffer.extend(samples);
```

```
while buffer.len() >= chunk_size { // chunk_size = 6 seconds worth
    let chunk: Vec<f32> = buffer.drain(..chunk_size).collect();
    let is_speech = vad.lock().unwrap().is_speech(&chunk)?;

    if is_speech > 0.5 {
        whisper.lock().unwrap().transcribe_chunk(&chunk, sample_rate)?;
    }
}
```

Step-by-Step:

REAL-TIME TRANSCRIPTION (Whisper Path)

STEP 1: Accumulate audio in buffer

```
buffer = [] (empty array)

Every 10ms: New audio arrives!
├ 10ms: buffer = [480 samples]
├ 20ms: buffer = [960 samples]
├ 30ms: buffer = [1440 samples]
└ ...
└ 6000ms: buffer = [288,000 samples]  READY!
```

chunk_size = sample_rate * 6 = 48000 * 6 = 288,000
└ 6 seconds of audio

↓

STEP 2: Extract chunk from buffer

```
let chunk = buffer.drain(..chunk_size).collect();

Before:
buffer = [288,000 samples, maybe more...]

drain() = Take and REMOVE first 288,000 samples

After:
chunk = [288,000 samples] ← Extracted
buffer = [remaining samples] ← What's left

Analogy: Scooping soup
Take a ladle (chunk) from the pot (buffer)
```

↓

STEP 3: Voice Activity Detection (VAD)

⌚ src-tauri/src/vad.rs (Line 78)

```
let is_speech = vad.is_speech(&chunk)?;
```

What happens:

1. Calculate RMS (Root Mean Square) = Loudness
$$\text{rms} = \sqrt{\frac{\sum(x^2)}{\text{count}}}$$

2. Compare to threshold (0.005)

- Too quiet? → 0.0 (silence)
- Very loud? → 1.0 (speech)
- In between? → 0.0-1.0 (probability)

Example:

- Silence: is_speech = 0.1
- Speech: is_speech = 0.85

Why? Skip transcribing silence (saves time/money!)

↓

STEP 4: Transcribe if speech detected

```
if is_speech > 0.5 { // More than 50% sure it's speech
    whisper.transcribe_chunk(&chunk, sample_rate)?;
}
```

If VAD says "YES, this is speech":

→ Send to Whisper for transcription

If VAD says "NO, this is silence":

→ Skip! Save compute. Don't bother the AI.

↓

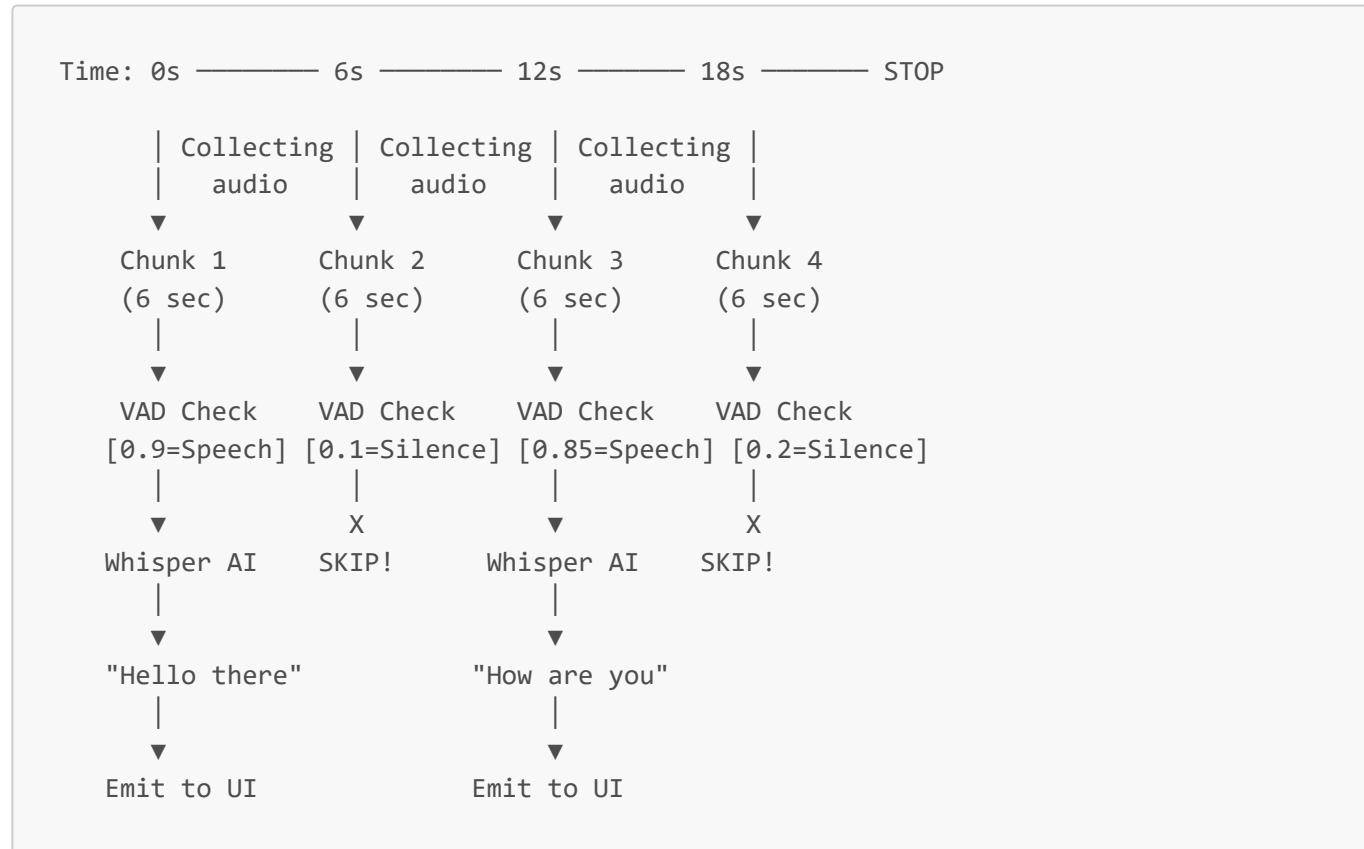
STEP 5: Whisper Transcription

⌚ src-tauri/src/whisper.rs (Line 345)

```
whisper.transcribe_chunk(&chunk, sample_rate)
```

(We'll detail this next!)

Visual Timeline:



⚡ How is Audio Fed into Whisper Model?

♀ Location: [src-tauri/src/whisper.rs \(Lines 345-458\)](#)

Now let's see what happens INSIDE the Whisper transcription function!

```
pub fn transcribe_chunk(  
    &mut self,  
    samples: &[f32],  
    input_sample_rate: u32,  
) -> Result<String, String>
```

Complete Process:

INPUT: chunk = [288,000 samples at 48kHz]

STEP 1: Resample Audio (Lines 357-391)

Why? Whisper ONLY works with 16kHz audio!
Your mic might record at 48kHz or 44.1kHz

```
if input_sample_rate != 16000 {  
    // Use resampler to convert  
    let resampler = SincFixedIn::new(  
        16000 / input_sample_rate, // Target ratio  
        ...  
    );  
    audio_data = resampler.process(&samples)?;  
}
```

Before: 288,000 samples @ 48kHz = 6 seconds
After: 96,000 samples @ 16kHz = 6 seconds

Analogy: Video Format Conversion
Converting 4K video → 1080p (same content, different size)

STEP 2: Create Whisper State (Lines 394-396)

```
let mut state = ctx.create_state()?;

What's a "state"?  
→ A temporary workspace for THIS transcription  
→ Holds intermediate calculations  
→ Gets thrown away after we're done
```

Analogy: Chef's Cutting Board
A clean surface to prepare THIS dish

STEP 3: Configure Parameters (Lines 400-415)

```
let mut params = FullParams::new(  
    SamplingStrategy::Greedy { best_of: 1 }  
);  
  
params.set_n_threads(4); // Use 4 CPU cores  
params.set_language(Some("en")); // English  
params.set_translate(false); // Don't translate  
  
// CONTEXT MAGIC! (Lines 413-415)  
if !self.last_transcript.is_empty() {  
    params.set_initial_prompt(&self.last_transcript);  
}
```

```
What's initial_prompt?  
→ Tell Whisper what was said BEFORE  
→ Helps maintain context across chunks  
→ Example: "The cat sat on the" → Next: "mat"
```

Analogy: Story Context
"Previously on Taurscribe: ..."



STEP 4: Run Whisper AI! (Lines 421-423)

```
state.full(params, &audio_data)?;
```

🚀 THIS IS WHERE THE MAGIC HAPPENS!

What happens inside (whisper-rs library):

1. Compute Mel Spectrogram (audio → visual representation)
2. Run through Encoder (understands audio features)
3. Run through Decoder (generates text tokens)
4. Beam search / Sampling (pick best words)

Time: 100-500ms on GPU, 1-3 seconds on CPU

Analogy: Oracle's Prophecy
Give the AI audio, it speaks back text!



STEP 5: Extract Text (Lines 426-432)

```
let num_segments = state.full_n_segments();  
let mut transcript = String::new();  
  
for i in 0..num_segments {  
    if let Some(segment) = state.get_segment(i) {  
        transcript.push_str(&segment.to_string());  
    }  
}
```

What's a segment?
→ Whisper breaks text into pieces (usually sentences)
→ Each segment has text + timestamp
→ We concatenate them all together

Example:
Segment 0: "Hello"

```
Segment 1: " there"
Segment 2: ", how are you?"
Combined: "Hello there, how are you?"
```



STEP 6: Update Context Memory (Lines 438-443)

```
if !final_text.is_empty() {
    if !self.last_transcript.is_empty() {
        self.last_transcript.push(' ');
    }
    self.last_transcript.push_str(&final_text);
}
```

Save this text for next chunk!
This accumulated text becomes the "initial_prompt"

Example Timeline:

```
Chunk 1: "Hello" → last_transcript = "Hello"
Chunk 2: "there" → last_transcript = "Hello there"
Chunk 3: "friend" → last_transcript = "Hello there friend"
```



STEP 7: Return Text (Line 457)

```
Ok(final_text)
```

Text travels back through the call stack...
→ transcribe_chunk() returns
→ Back to recording.rs
→ Emit to frontend!

OUTPUT: "Hello there, how are you?"

Visual Architecture of Whisper Model:

```
WHISPER MODEL
(Inside whisper-rs library)
```

```
INPUT: audio_data [96,000 samples @ 16kHz]
```



🐦 How is Audio Sent to Parakeet?

Parakeet is **streaming-only** - no final pass needed!

⌚ **Location:** `src-tauri/src/commands/recording.rs` (**Lines 140-182**)

```

} else { // Parakeet path
    buffer.extend(samples);

    let parakeet_chunk_size = (sample_rate as f32 * 1.12) as usize;

    while buffer.len() >= parakeet_chunk_size {
        let chunk = buffer.drain(..parakeet_chunk_size).collect();
    }
}
  
```

```
    parakeet_manager.transcribe_chunk(&chunk, sample_rate)?;  
}  
}
```

Breaking It Down:

REAL-TIME TRANSCRIPTION (Parakeet Path)

STEP 1: Accumulate audio (same as Whisper)

```
buffer = [] (empty)  
Every 10ms: buffer.extend(new_samples)  
  
BUT! Chunk size is different:  
parakeet_chunk_size = sample_rate * 1.12  
= 48000 * 1.12  
= 53,760 samples  
= ~1.12 seconds of audio
```

Why 1.12 seconds?

- Parakeet is FASTER than Whisper
- Can process smaller chunks quickly
- More responsive (updates more frequently)

STEP 2: Extract chunk (same as Whisper)

```
let chunk = buffer.drain(..parakeet_chunk_size).collect();  
  
chunk = [53,760 samples] @ 48kHz = 1.12 seconds
```

STEP 3: NO VAD CHECK!

Parakeet path SKIPS voice activity detection!

Why?

- └ Parakeet is very fast (can handle all audio)
- └ Better continuity (no gaps from skipping silence)
- └ Streaming models need consistent input

STEP 4: Transcribe with Parakeet

```
parakeet_manager.transcribe_chunk(&chunk, sample_rate)?;
```

♀ src-tauri/src/parakeet.rs (Line 528)

(We'll detail this next!)

STEP 5: Accumulate in Session Transcript

```
let mut session = session_transcript.lock().unwrap();
if !session.is_empty() {
    session.push(' '); // Add space
}
session.push_str(transcript.trim());
```

Why? Parakeet outputs INCREMENTAL text
Each chunk adds to the full transcript

Example:

```
Chunk 1: "Hello" → session = "Hello"  
Chunk 2: "there" → session = "Hello there"  
Chunk 3: "friend" → session = "Hello there friend"
```

Comparison: Whisper vs Parakeet Chunking

WHISPER (6 second chunks):

Time: 0 ————— 6 ————— 12 ————— 18 ————— 24 seconds

|————|————|————|————|

Chunk 1 Chunk 2 Chunk 3 Chunk 4

Less frequent updates, but more accurate

PARAKEET (1.12 second chunks):

Time: 0—1.12—2.24—3.36—4.48—5.60 seconds
| = | = | = | = | = | = | = | = |

C1 C2 C3 C4 C5 C6 C7 C8 C9 C10

Very frequent updates, streaming feel!

⚡ How is Audio Fed into Parakeet Model?

⌚ **Location:** [src-tauri/src/parakeet.rs](#) (**Lines 528-613**)

```
pub fn transcribe_chunk(  
    &mut self,  
    samples: &[f32],  
    sample_rate: u32,  
) -> Result<String, String>
```

Complete Process:

INPUT: chunk = [53,760 samples at 48kHz]

STEP 1: Resample to 16kHz (Lines 534-567)

Same as Whisper! Parakeet also needs 16kHz.

Before: 53,760 samples @ 48kHz = 1.12 seconds
After: 17,920 samples @ 16kHz = 1.12 seconds



STEP 2: Match with correct model type (Lines 569-609)

Parakeet has 4 different model architectures:

1. NEMOTRON (Streaming RNN-T)
 - Best for real-time
 - Has internal state (memory)
 - Process in tiny 560ms chunks

2. CTC (Connectionist Temporal Classification)
 - Fast offline processing
 - No state (stateless)
 - Process whole chunk at once

3. EOU (End of Utterance)
 → Detects sentence boundaries
 → Process in 160ms micro-chunks

4. TDT (Token-and-Duration Transducer)
 → Includes timing information
 → Process whole chunk

```
match model {
    LoadedModel::Nemotron(m) => { ... }
    LoadedModel::Ctc(m) => { ... }
    LoadedModel::Eou(m) => { ... }
    LoadedModel::Tdt(m) => { ... }
}
```

PATH A: NEMOTRON (Most common for streaming)

Lines 571-582

```
LoadedModel::Nemotron(m) => {
    let mut transcript = String::new();
    const CHUNK_SIZE: usize = 8960; // 560ms @ 16kHz

    for chunk in audio.chunks(CHUNK_SIZE) {
        let mut chunk_vec = chunk.to_vec();
        if chunk_vec.len() < CHUNK_SIZE {
            chunk_vec.resize(CHUNK_SIZE, 0.0); // Pad!
        }
        transcript.push_str(
            &m.transcribe_chunk(&chunk_vec)?
        );
    }
    Ok(transcript)
}
```

What's happening:

1. Break 1.12s audio into even SMALLER pieces (560ms)
2. Nemotron processes each 560ms chunk
3. Concatenate results

Why so small?

- Nemotron is an RNN-T (Recurrent Neural Network)
- Maintains hidden state across chunks

→ Very responsive (updates every 560ms!)

Visual:

1.12s audio:

[—————]

[————][————]

560ms 560ms

Chunk1 Chunk2

|

|

"Hello" " there"

Combined: "Hello there"

INSIDE Nemotron Model (parakeet-rs library)

m.transcribe_chunk(&chunk)

Architecture:

INPUT: [8960 samples] @ 16kHz = 560ms

ENCODER (ONNX model: encoder.onnx)

Processes audio features

Outputs: Encoded representations

Uses: Recurrent layers (LSTM/GRU)

DECODER + JOINT NETWORK

(ONNX model: decoder_joint.onnx)

Loop:

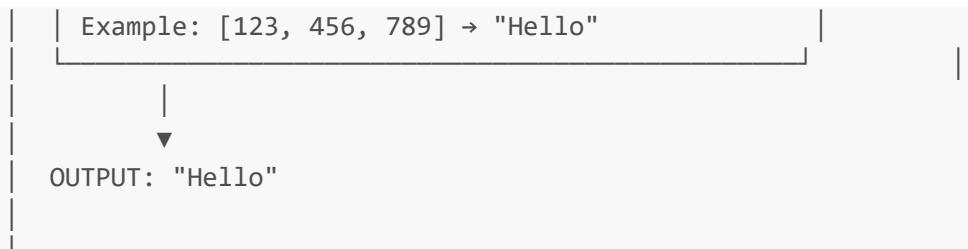
1. Predict next token
2. Update internal state
3. Repeat until end-of-utterance

Internal State = "Memory" of previous words

This is WHY we can call .reset() to clear it!

TOKENIZER (tokenizer.model - SentencePiece)

Token IDs → Text



OUTPUT: "Hello there" (or whatever was said)

Key Difference: Nemotron has STATE!

NEMOTRON (Stateful):

```
Chunk 1: "Hello"    ↗ Internal State Updated
          ↘
Chunk 2: " there"  ↗ (Remembers "Hello")
          ↘
          ↗ Better context across chunks!
```

CTC/TDT/EOU (Stateless):

```
Chunk 1: "Hello" → Process → Output
          ↓
          Forget everything
          ↓
Chunk 2: "there" → Process → Output
Each chunk is independent!
```

🖨 How is Text Outputted?

Text gets back to the UI through **Tauri Events!**

⌚ Location: [src-tauri/src/commands/recording.rs](#) (Lines 121-128 for Whisper, 163-170 for Parakeet)

```
// After transcription finishes:
app_clone.emit(
  "transcription-chunk",
  TranscriptionChunk {
    text: transcript,
    processing_time_ms: elapsed,
    method: "Whisper".to_string(),
  },
)?;
```

Complete Flow:

```
RUST BACKEND (Transcriber Thread)
```

1. AI finishes transcription
transcript = "Hello there"
2. Create event payload
TranscriptionChunk {
 text: "Hello there",
 processing_time_ms: 250,
 method: "Whisper",
}
3. Emit event to frontend
app_handle.emit("transcription-chunk", payload);

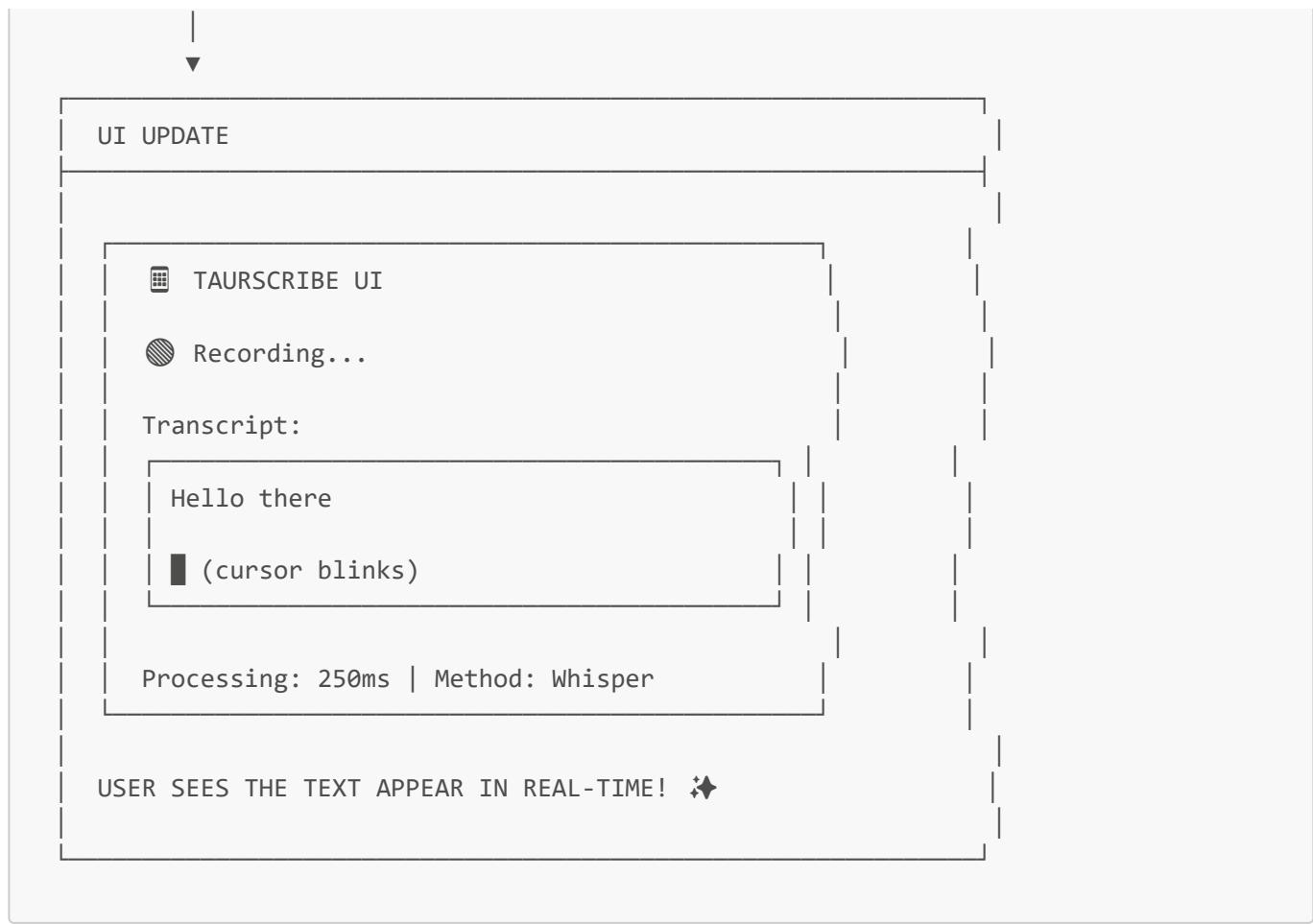
What's emit()?
→ Tauri's event system
→ Broadcasts a message to ALL listeners
→ Like shouting in a room

(Event travels through Tauri bridge...)

```
FRONTEND (React) - Event Listener  
⌚ src/App.tsx (somewhere in useEffect)
```

```
useEffect(() => {  
    const unlisten = listen(  
        "transcription-chunk",  
        (event) => {  
            const chunk = event.payload;  
            setLiveTranscript(prev => prev + chunk.text);  
        }  
    );  
    return () => unlisten();  
}, []);
```

What happens:
1. listen() registers an event handler
2. When "transcription-chunk" event arrives...
3. Callback function runs
4. Updates React state (setLiveTranscript)
5. React re-renders UI with new text!

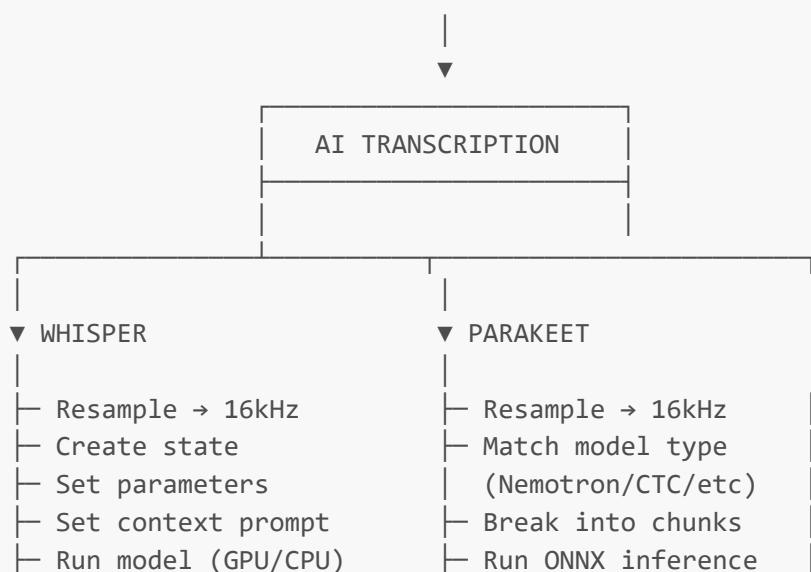
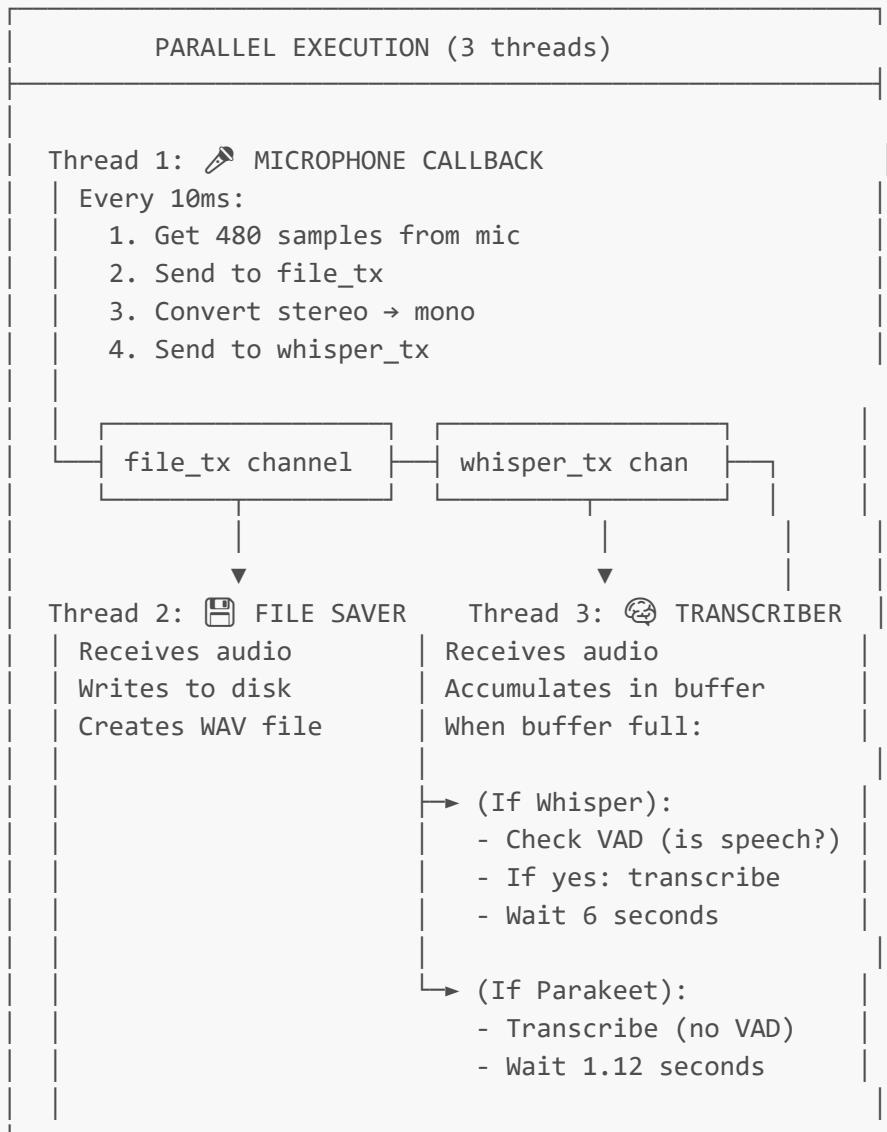


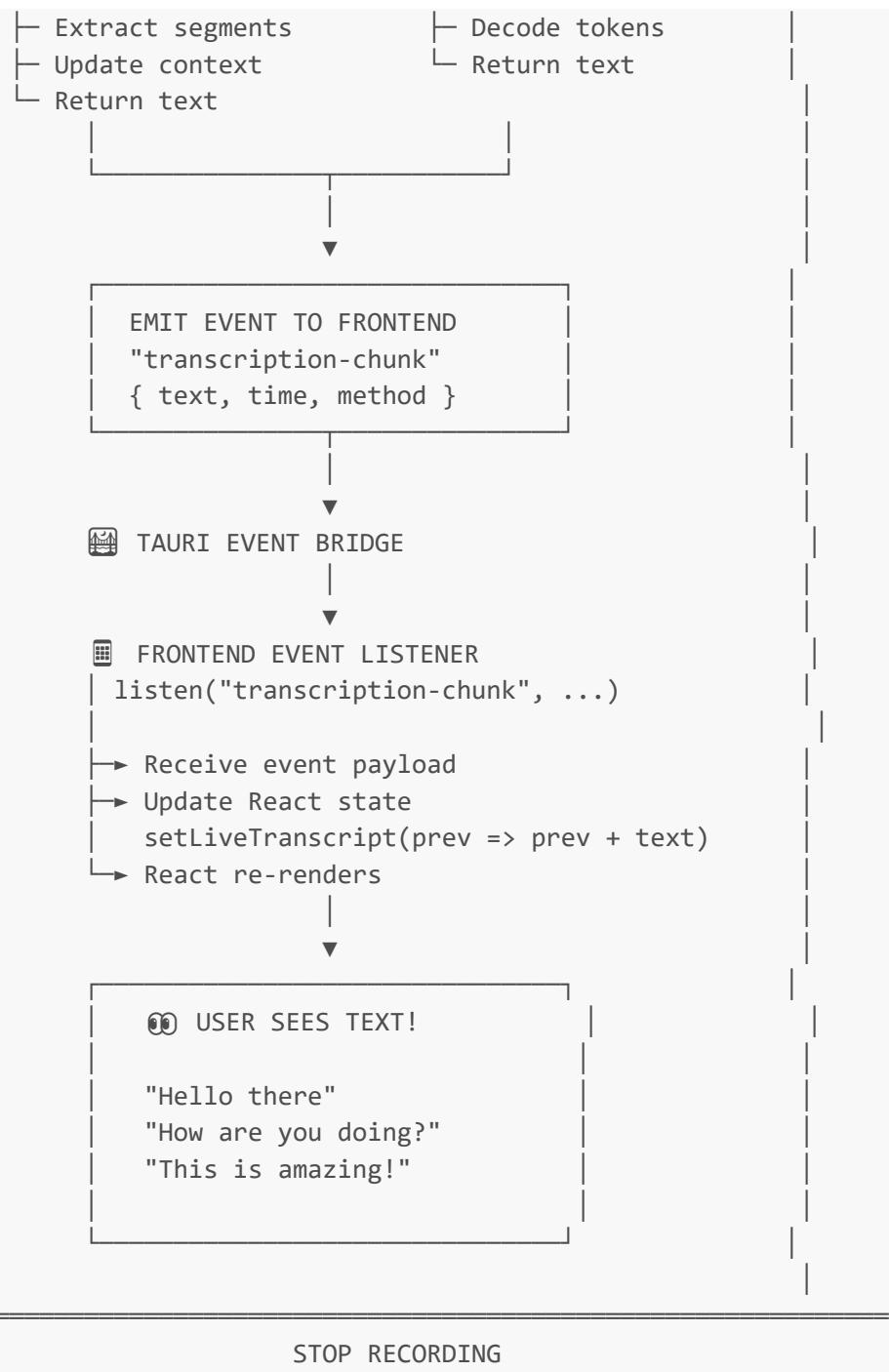
🎬 Complete Visual Flow: Start to Finish

Here's the ENTIRE journey from button click to seeing text:



- STEP 4: Create channels (file_tx/rx, whisper_tx/rx)
 - STEP 5: Spawn File Saver Thread 📁
 - STEP 6: Spawn Transcriber Thread 🎧
 - STEP 7: Start microphone stream ▶
- Microphone now captures audio every 10ms!





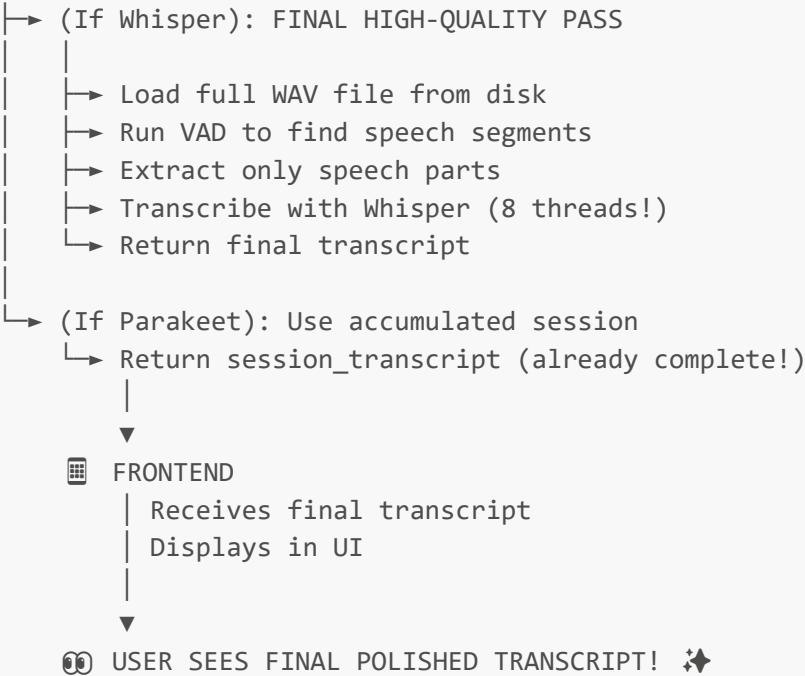
```

USER
Clicks [ ] Stop Recording

FRONTEND
await invoke("stop_recording")

RUST BACKEND (src-tauri/src/commands/recording.rs:276)
stop_recording() function

    Drop stream (stops microphone)
    Drop channels (closes pipes)
    Wait 500ms (let threads finish)
  
```



🎓 Key Takeaways for Beginners

1. Threads & Parallelism

Think of threads like workers in a restaurant:

- Worker 1 (Mic): Takes orders constantly
- Worker 2 (File): Writes down every order
- Worker 3 (AI): Translates orders for the chef

All working at the SAME TIME!

2. Channels (Pipes)

Channels are like conveyor belts:

- `send()` = Put item on belt
- `recv()` = Take item off belt
- Data flows from one thread to another safely!

3. Buffering

We collect audio in a "buffer" (basket) until we have enough:

- Too small: Waste time processing tiny pieces
- Too big: User waits forever for results
- Just right: 6 seconds for Whisper, 1.12s for Parakeet

4. Sample Rates

Audio frequency (like video frame rate):

- 48kHz = 48,000 samples per second (high quality)
- 16kHz = 16,000 samples per second (AI standard)
- Resampling = Converting between rates

5. Stereo vs Mono

- Stereo = 2 channels (Left + Right ears)
- Mono = 1 channel (averaged)
- AI models need mono!

6. Events

Rust → Frontend communication:

- `emit()` = Shout a message
- `listen()` = Ear listening for messages
- Real-time updates without polling!

Summary Table: Where Everything Happens

Task	File	Function	Line
Start button click	<code>src/App.tsx</code>	Button onClick	~168
Start recording command	<code>src-tauri/src/commands/recording.rs</code>	<code>start_recording()</code>	13-272
Setup microphone	↑ Same	↑ Same	17-23
Create channels	↑ Same	↑ Same	54-55
File saver thread	↑ Same	↑ Same	62-72
Transcriber thread	↑ Same	↑ Same	82-235
Audio callback	↑ Same	↑ Same	243-254
Whisper real-time path	↑ Same	Transcriber thread	94-139
Parakeet real-time path	↑ Same	Transcriber thread	140-182
VAD check	<code>src-tauri/src/vad.rs</code>	<code>is_speech()</code>	78-98

Task	File	Function	Line
Whisper transcription	<code>src-tauri/src/whisper.rs</code>	<code>transcribe_chunk()</code>	345-458
Whisper model inference	↑ Same (whisper-rs lib)	<code>state.full()</code>	421-423
Parakeet transcription	<code>src-tauri/src/parakeet.rs</code>	<code>transcribe_chunk()</code>	528-613
Nemotron inference	↑ Same (parakeet-rs lib)	<code>m.transcribe_chunk()</code>	579
Emit event to frontend	<code>src-tauri/src/commands/recording.rs</code>	<code>app_clone.emit()</code>	121-128
Stop recording command	↑ Same	<code>stop_recording()</code>	276-344
Whisper final pass	↑ Same	↑ Same	306-326
Load audio file	<code>src-tauri/src/whisper.rs</code>	<code>load_audio()</code>	707-779
Final transcription	↑ Same	<code>transcribe_audio_data()</code>	647-703

🎉 You Now Understand the Complete Audio Pipeline!

From clicking a button to seeing text appear on screen, you now know:

- Where every piece of code lives
- How audio flows through the system
- How Whisper processes audio (chunks + final pass)
- How Parakeet streams audio (smaller chunks)
- How text gets back to the UI
- Why we use threads, channels, and buffers
- The difference between real-time and final transcription

You're ready to modify, debug, and extend the audio system! 💡

Done!

🚀 Major Update: Grammar Correction with Gemma LLM (Commits `f54196f` → `b0aeb9d`)

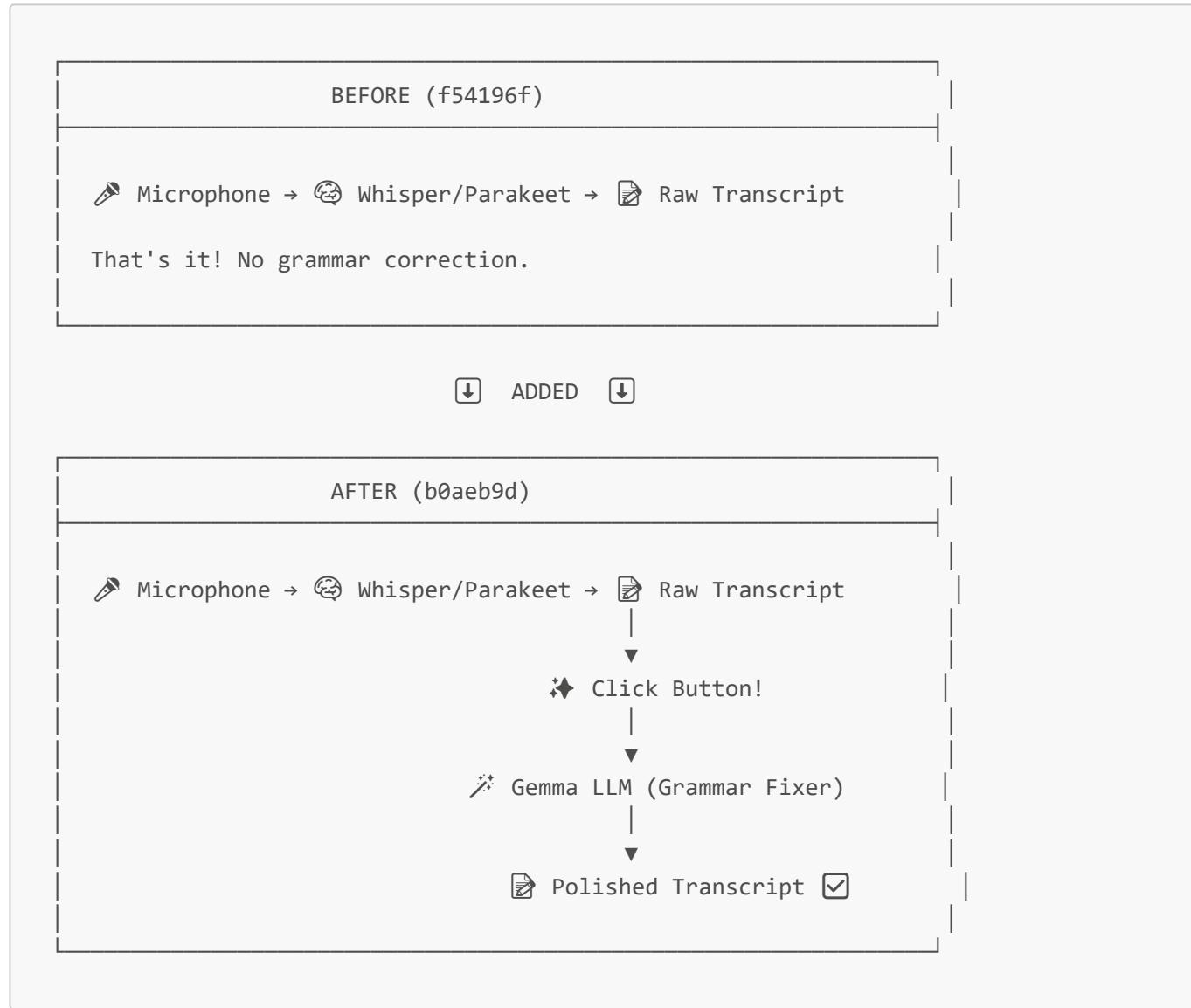
📋 What Changed?

Between commit `f54196f` (old) and `b0aeb9d` (latest), we added **AI-powered grammar correction** to Taurscribe! Now after you transcribe your voice, you can click a button to automatically fix grammar mistakes

using a local Large Language Model (LLM).

Think of it like: Adding a professional editor to your restaurant who instantly polishes your transcript!

⌚ Overview of Changes

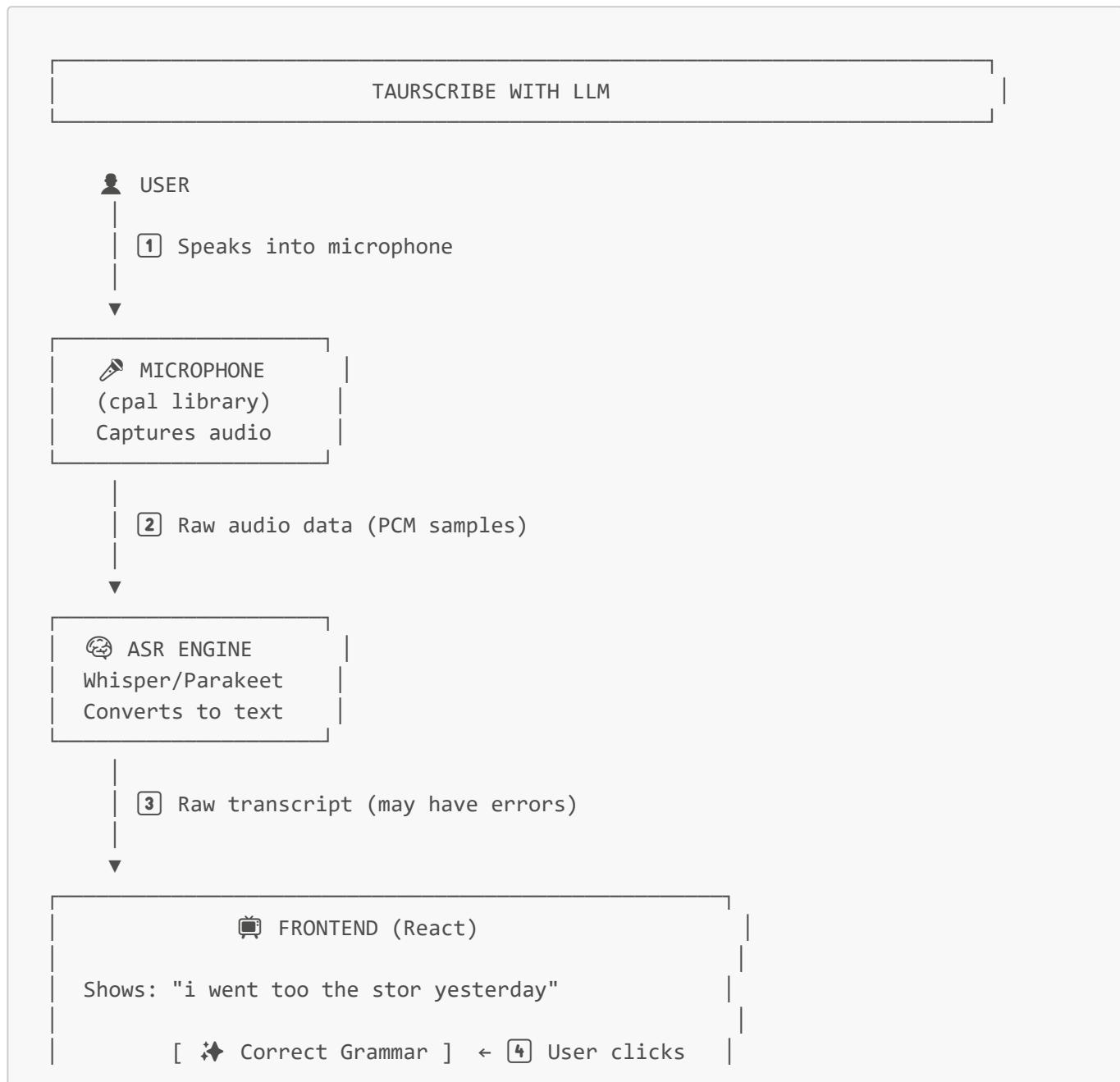


📊 File Changes Summary

File	Status	Purpose
src-tauri/src/llm.rs	NEW	Core LLM engine using Candle framework
src-tauri/src/commands/llm.rs	NEW	Command handlers for frontend-backend communication
src-tauri/src/state.rs	MODIFIED	Added LLM storage to global state
src-tauri/src/lib.rs	MODIFIED	Registered new LLM commands

File	Status	Purpose
src-tauri/src/commands/mod.rs	MODIFIED	Exported LLM commands
src-tauri/Cargo.toml	MODIFIED	Added Candle ML framework dependencies
src-tauri/.cargo/config.toml	MODIFIED	Fixed linking issues for Windows
src/App.tsx	MODIFIED	Added "Correct Grammar" button
src/App.css	MODIFIED	Styled the correction button

Diagram: The Complete Architecture Flow



5 invoke("correct_text", { text: "..." })
(Tauri bridge sends request to Rust backend)

6 RUST BACKEND (commands/llm.rs)
correct_text() function receives request
"Someone wants to fix this text!"

7 Access global state

8 GLOBAL STATE (state.rs)
pub llm: Arc<Mutex<Option<LLMEngine>>>
└ Thread-safe box holding LLM

9 Lock mutex, get LLM engine

10 LLM ENGINE (llm.rs)
Powered by: Candle + Gemma 3 Model
Model: GRMR-V3-G1B-Q4_K_M.gguf (1 billion params)

11 Format prompt for grammar correction

12 Formatted Prompt:
<bos>text
i went too the stor yesterday
corrected
(Model learns to complete this pattern)

13 Tokenize (text → numbers)

14 Token IDs: [1, 1234, 5678, 9012, ...]

15 Run through neural network



NEW NEW FILE: [src-tauri/src/llm.rs](#)

This is the **heart of the grammar correction system**. It's like the master chef who specializes in making dishes (text) perfect!

❖ Structure Overview

```
pub struct LLMEngine {
    model: model::ModelWeights,           // The "brain" - Gemma neural network
    tokenizer: Tokenizer,                 // Translator (text ↔ numbers)
    device: Device,                      // Where to compute (GPU/CPU)
    logits_processor: LogitsProcessor // Decision maker (sampling)
}
```

Analogy: Think of **LLMEngine** as a **magic translation machine**:

⚡ THE GRAMMAR CORRECTION MACHINE

- ❑ tokenizer (Dictionary)
 - "Converts words to numbers and back"
 - "hello" becomes [1234]
 - [5678] becomes "world"

- ❑ model (The Brain)
 - "1 billion parameters (memories/patterns)"
 - Learned from millions of text examples
 - Knows grammar rules implicitly

- ❑ device (The Workstation)
 - "Where the work happens"
 - GPU (Graphics Card) = Fast! ⚡
 - CPU (Regular Processor) = Slower 🐢

- ❑ logits_processor (The Decision Maker)
 - "Picks the best word from possibilities"
 - Temperature 0.7 = Some creativity
 - Top-P 0.95 = Focus on best options

🔧 Function: `LLMEngine::new()` - Building the Machine

This function **loads** the LLM into memory. It's like assembling a complex machine before using it.

```
pub fn new() -> Result<Self> {
    // 1. Define where the model files live
    let base_path = PathBuf::from(
        r"c:\Users\abdul\OneDrive\Desktop\Taurscribe\taurscribe-
        runtime\models\GRMR-V3-G1B-GGUF",
    );
    let model_path = base_path.join("GRMR-V3-G1B-Q4_K_M.gguf");
    let tokenizer_path = base_path.join("tokenizer.json");

    // 2. Check if files exist (safety check)
    if !model_path.exists() {
        return Err(Error::msg("Model file not found!"));
    }

    // 3. Select device (try GPU first, fallback to CPU)
    let device = Device::new_cuda(0).unwrap_or(Device::Cpu);

    // 4. Load tokenizer (the translator)
    let tokenizer = Tokenizer::from_file(&tokenizer_path)?;

    // 5. Load model weights (the brain)
    let mut file = std::fs::File::open(&model_path)?;
    let content = gguf_file::Content::read(&mut file)?;
    let model = model::ModelWeights::from_gguf(content, &mut file, &device)?;
}
```

```
// 6. Create the decision maker
let logits_processor = LogitsProcessor::new(1337, Some(0.7), Some(0.95));

Ok(Self { model, tokenizer, device, logits_processor })
}
```

Step-by-Step Analogy: Building a Coffee Machine ☕

BUILDING THE GRAMMAR CORRECTION MACHINE

Step 1: Find the blueprint (model file path)

- 📁 "Where are the instructions?"
- Check c:\...\GRMR-V3-G1B-Q4_K_M.gguf

Step 2: Verify files exist

- 🔍 "Do we have everything we need?"
- Yes? Continue. No? Error!

Step 3: Choose where to work

- 💻 "Do we have a fast GPU?"
- Try CUDA (NVIDIA GPU): Success? Use it! ⚡
- No GPU available? Use CPU instead 🖥

Step 4: Load the dictionary

- 📖 "Load tokenizer.json"
- Now we can translate words ↔ numbers

Step 5: Load the brain

- 🧠 "Open the 1GB model file and load into memory"
- This takes 2-3 seconds
- Contains 1 billion learned patterns!

Step 6: Prepare the decision maker

- 🎲 "Set up sampling rules"
- Seed: 1337 (for reproducibility)
- Temperature: 0.7 (controlled creativity)
- Top-P: 0.95 (focus on best options)

- Machine ready! Return it to the caller

Key Rust Concepts:

1. **Result<Self>**: This function can succeed (return `Ok(LLMEngine)`) or fail (return `Err(Error)`).
 - Think: "This might not work, so we need a backup plan."
2. **? operator**: The magic error handler!

- If something fails, immediately return the error to the caller.
- Example: `let tokenizer = Tokenizer::from_file(&path)?;`
- If `from_file` fails, the `?` says "stop here and return the error."

3. `unwrap_or`: The fallback operator

- `Device::new_cuda(0).unwrap_or(Device::Cpu)`
- Try CUDA first. If it fails, use CPU instead.

⚡ Function: `LLMEngine::run()` - The Magic Happens

This function takes messy text and returns corrected text!

```
pub fn run(&mut self, prompt: &str) -> Result<String> {
    // 1. Format the prompt using a special template
    let formatted_prompt = format!("<bos>text\n{}\\ncorrected\\n", prompt.trim());

    // 2. Convert text to token IDs (numbers)
    let tokens = self.tokenizer
        .encode(formatted_prompt, true)?
        .get_ids()
        .to_vec();

    // 3. Create a tensor (multidimensional array) for the model
    let input = Tensor::new(tokens.as_slice(), &self.device)?;

    // 4. Run the model! (The magic neural network computation)
    let logits = self.model.forward(&input, 0)?;

    // 5. Get the predictions for the next token
    let (seq_len, _vocab_size) = logits.dims2()?;
    let last_logits = logits.get(seq_len - 1)?;

    // 6. Sample the next token (pick the best one)
    let next_token = self.logits_processor.sample(&last_logits)?;

    // 7. Convert token ID back to text
    let decoded = self.tokenizer.decode(&[next_token], true)?;

    Ok(decoded)
}
```

Analogy: The Grammar Correction Assembly Line 🚧

INPUT: "i went too the stor yesterday"



STEP 1: Format Prompt (Add Magic Instructions)

```
Before: "i went too the stor yesterday"  
After:  "<bos>text  
        i went too the stor yesterday  
        corrected  
        "  
  
Why? The model was trained to complete this pattern!
```

It knows: "Oh, I need to fix this text now"

STEP 2: Tokenize (Text → Numbers)

```
"<bos>" → [1]  
"text"   → [1234]  
"i"      → [5678]  
"went"   → [9012]  
"too"    → [3456]  
...  
  
Result: [1, 1234, 5678, 9012, 3456, ...]
```

Why numbers? Computers can't understand words directly. They need numbers to do math!

STEP 3: Create Tensor (Organize the Numbers)

```
Tensor = Fancy word for "array of numbers"  
  
Shape: [sequence_length]  
Example: [1, 1234, 5678, 9012, 3456, 7890, ...]  
  
Device: GPU or CPU  
(Moves data to the right place for computation)
```

STEP 4: Model Forward Pass (THE MAGIC! ✨)

```
self.model.forward(&input, 0)
```

What happens inside:

1. Embeddings: Each token → 576 numbers
2. Self-Attention (×30 layers):
 - "Which words relate to each other?"
 - "too" looks at "went" for context
3. Feed-Forward Networks:
 - Transform representations

```
- 576 → 1536 → 576 dimensions
4. Output: Predictions for EVERY position

Result: "logits" = Raw predictions
Shape: [sequence_length, 49152]
└ 49,152 possible tokens in vocabulary

⚠ This is where 1 BILLION PARAMETERS work!
(Billions of multiply-add operations)
```

STEP 5: Extract Last Token's Predictions

```
logits shape: [10, 49152]
(10 tokens in, 49152 possible outputs each)
```

```
We want: logits[9] (the LAST position)
Shape: [49152]
```

This represents: "What should come next?"

Example values:

- Token 3456 ("I"): probability 0.85 ★
- Token 7890 ("i"): probability 0.05
- Token 1111 ("We"): probability 0.02
- ... 49,149 other possibilities

STEP 6: Sample Next Token (Make a Decision)

```
self.logits_processor.sample(&last_logits)
```

What it does:

1. Apply temperature (0.7):
 - Makes the distribution more/less uniform
 - Lower = more confident, higher = more random
2. Apply Top-P (0.95):
 - Only consider top 95% probability mass
 - Ignores unlikely options
3. Sample randomly from remaining options:
 - Usually picks the most likely token
 - But has a small chance to pick others

Selected: Token 3456 ("I")

```
STEP 7: Decode Token → Text

self.tokenizer.decode(&[3456], true)

Looks up token 3456 in dictionary:
3456 → "I"

Result: "I"
```

↓
OUTPUT: "I"

NOTE: This simplified version only generates ONE token!
 For full sentence correction, you'd loop and generate
 more tokens until you hit a stop token.

Important Rust Concepts:

1. **&mut self**: The function can modify the **LLMEngine** instance
 - The model's internal state might change during inference
2. **&str**: Borrowed string (no ownership transfer)
 - We just read the prompt, we don't need to own it
3. **.trim()**: Removes whitespace from start/end of string
4. **Tensor**: A multi-dimensional array (from Candle library)
 - Think: Excel spreadsheet, but with more dimensions!

NEW NEW FILE: **src-tauri/src/commands/llm.rs**

This file contains **4 command functions** that connect the frontend (React) to the backend (Rust LLM engine).

Analogy: These are the **waiters** who take orders from customers and bring them to the chef!

COMMAND FUNCTIONS (Waiters)

- | | |
|-------------------------------------|-------------------------|
| 1. <code>init_llm()</code> | - "Load the chef!" |
| 2. <code>check_llm_status()</code> | - "Is the chef ready?" |
| 3. <code>run_llm_inference()</code> | - "Chef, make this!" |
| 4. <code>correct_text()</code> | - "Fix grammar please!" |

🔧 Command 1: `init_llm()` - Start the Engine

```
#[tauri::command]
pub async fn init_llm(state: State<'_, AudioState>) -> Result<String, String> {
    println!("[COMMAND] init_llm requested");

    // Check if already loaded
    {
        let llm_guard = state.llm.lock().unwrap();
        if llm_guard.is_some() {
            return Ok("LLM already initialized".to_string());
        }
    }

    // Load in a blocking task (heavy operation)
    let result = tauri::async_runtime::spawn_blocking(move || LLMEngine::new())
        .await
        .map_err(|e| format!("JoinError: {}", e))?;

    match result {
        Ok(engine) => {
            let mut llm_guard = state.llm.lock().unwrap();
            *llm_guard = Some(engine);
            println!("[SUCCESS] Gemma LLM initialized!");
            Ok("Gemma LLM initialized successfully".to_string())
        }
        Err(e) => {
            eprintln!("[ERROR] Failed to load LLM: {}", e);
            Err(format!("Failed to load LLM: {}", e))
        }
    }
}
```

What's Happening: The Restaurant Analogy 🍽️

STEP 1: Customer asks "Can I get the special chef?"
(Frontend calls `init_llm()`)



STEP 2: Check if chef is already working

```
{
    let llm_guard = state.llm.lock().unwrap();
    if llm_guard.is_some() { return "Already here"; }
}
```

Why the curly braces {}?

- Creates a temporary scope
- Lock is released when scope ends
- Other threads can use `state.llm` after this

↓

STEP 3: Load the chef (heavy operation!)

```
spawn_blocking(move || LLMEngine::new())
```

Why blocking?

- Loading takes 2-3 seconds
- Would freeze the app if done on main thread
- `spawn_blocking` runs it on a separate thread

Analogy: Send someone to pick up the chef while customers continue ordering food

↓

STEP 4: Wait for loading to finish

```
.await - "Wait for the chef to arrive"
```

Meanwhile, other tasks can run! (Async magic ♦)

↓

STEP 5: Check result

```
match result {  
    Ok(engine) => {  
        // Chef arrived! Store him in the kitchen  
        let mut llm_guard = state.llm.lock().unwrap();  
        *llm_guard = Some(engine);  
        return Ok("Ready!");  
    }  
    Err(e) => {  
        // Chef couldn't come, explain why  
        return Err(format!("Failed: {}", e));  
    }  
}
```

Key Rust Concepts:

1. `#[tauri::command]`: Magic annotation that makes this function callable from JavaScript!
2. `async`: This function can wait without blocking

- Like a waiter who can serve other tables while waiting for food

3. `State<'_, AudioState>`: Access to global app state

- `'_` is a lifetime (compiler figures it out automatically)
- Think: "Give me access to the shared kitchen"

4. `lock().unwrap()`: Thread-safe access pattern

- `lock()`: "Wait until nobody else is using this"
- `unwrap()`: "I'm confident this will work, panic if it doesn't"

5. `spawn_blocking`: Run heavy work on a separate thread

- Prevents blocking the async runtime

6. `*llm_guard = Some(engine)`: Dereference and assign

- `llm_guard` is a MutexGuard (smart pointer)
- `*` dereferences it to access the inner value
- `Some(engine)` wraps the engine in an Option

🔧 Command 2: `check_llm_status()` - Is the Chef Ready?

```
#[tauri::command]
pub fn check_llm_status(state: State<'_, AudioState>) -> bool {
    let llm_guard = state.llm.lock().unwrap();
    llm_guard.is_some()
}
```

Simple! Just checks if the LLM is loaded.

```
Frontend: "Is the chef here?"
Backend: *Looks in kitchen*
        llm is Some(engine) → "Yes!" → return true
        llm is None          → "Nope!" → return false
```

Rust Concepts:

- `is_some()`: Checks if an `Option` contains a value
 - `Option::Some(x) → true`
 - `Option::None → false`

🔧 Command 3: `correct_text()` - Fix My Grammar!

This is the **main function** users interact with!

```

#[tauri::command]
pub async fn correct_text(
    state: State<'_, AudioState>,
    text: String
) -> Result<String, String> {
    println!("[LLM] correct_text request received. Input length: {}", text.len());

    let llm_handle = state.llm.clone();
    let prompt = text.clone();

    let output = tauri::async_runtime::spawn_blocking(move || {
        let mut llm_guard = llm_handle.lock().unwrap();
        if let Some(engine) = llm_guard.as_mut() {
            println!("[LLM] Running inference on text: '{}'", prompt.trim());
            engine.run(&prompt).map_err(|e| e.to_string())
        } else {
            eprintln!("[LLM] Error: Engine not initialized");
            Err("LLM not initialized. Please load Gemma first.".to_string())
        }
    })
    .await
    .map_err(|e| format!("Join Error: {}", e))??;

    println!("[LLM] Inference finished. Output length: {}", output.len());
    Ok(output)
}

```

The Complete Journey: 🛡️

👤 USER: Clicks "◆ Correct Grammar" button
 Text: "i went too the stor yesterday"

📺 FRONTEND (React/TypeScript):

```
const corrected = await invoke("correct_text", {
    text: "i went too the stor yesterday"
});
```

Sends request across Tauri bridge...

🦀 BACKEND: correct_text() receives request

Parameters:

- state: Access to global app state

```
- text: "i went too the stor yesterday"
```

STEP 1: Clone handles for thread safety

```
let llm_handle = state.llm.clone();
let prompt = text.clone();

Why clone?
- spawn_blocking moves values into new thread
- Arc::clone() is cheap (just copies a pointer)
- String::clone() copies text (needed for move)
```

STEP 2: Run inference on background thread

```
spawn_blocking(move || {
    // This code runs on a separate thread!
    let mut llm_guard = llm_handle.lock().unwrap();
    if let Some(engine) = llm_guard.as_mut() {
        engine.run(&prompt)
    } else {
        Err("LLM not initialized")
    }
})
```

Why background thread?

- LLM inference takes 50-200ms
- Would block async runtime otherwise
- App stays responsive during inference

STEP 3: Lock and access LLM engine

```
let mut llm_guard = llm_handle.lock().unwrap();
```

What happens:

1. Acquire mutex lock (wait if someone else using)
2. Get mutable access to Option<LLMEngine>
3. Now we can modify the engine (it's stateful)

STEP 4: Check if engine exists

```
if let Some(engine) = llm_guard.as_mut() {
```

```
"as_mut()" = Get mutable reference to inner value  
  
Option<LLMEngine>  
|  
└ Some(engine) → Got it! Use it ✓  
└ None → Not loaded! Error ✗
```

STEP 5: Run the LLM! 🚀

```
engine.run(&prompt)  
  
This calls LLMEngine::run() we saw earlier!  
- Format prompt  
- Tokenize  
- Run neural network  
- Sample token  
- Decode  
  
Result: "I went to the store yesterday"
```

STEP 6: Wait for thread to finish

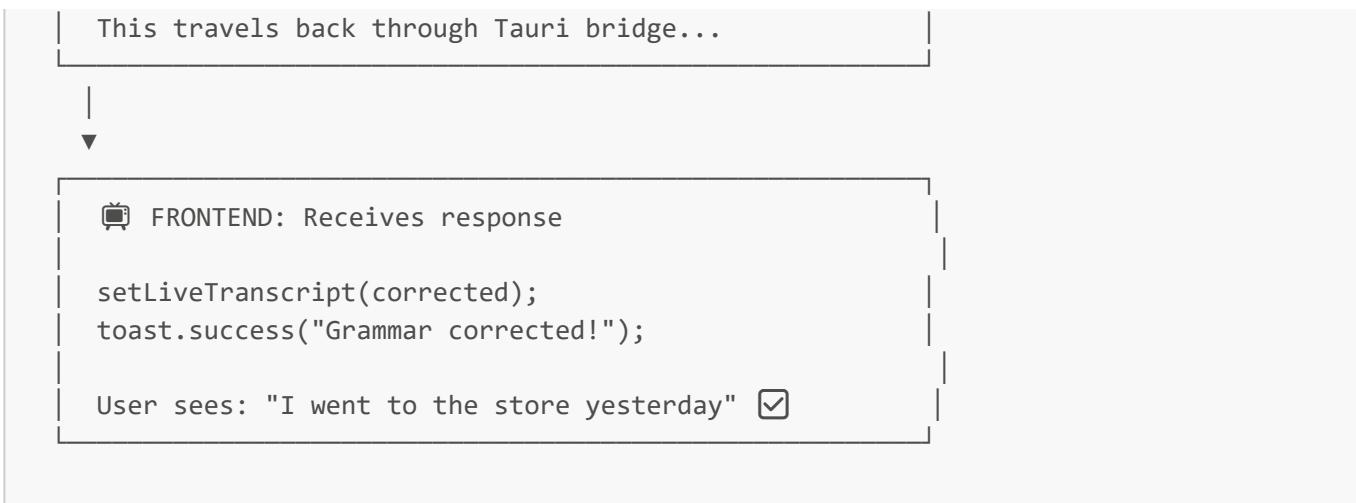
```
.await  
  
The spawn_blocking task completes  
Returns: Result<Result<String, String>, JoinError>  
└ Inner result from engine.run()  
└ Outer result from thread join
```

STEP 7: Handle errors with ?? (double question mark!)

```
.map_err(|e| format!("Join Error: {}", e))??  
  
First ? - Handle JoinError (if thread panicked)  
Second ? - Handle engine.run() error (if LLM failed)  
  
If both succeed: unwrap the inner Result<String>
```

STEP 8: Return corrected text

```
Ok("I went to the store yesterday")
```



Key Rust Concepts:

1. `if let Some(engine) = llm_guard.as_mut()`: Pattern matching!
 - If the Option contains a value, extract it
 - Otherwise, skip this block
2. `?? (double question mark)`: Error propagation twice
 - First ?: Convert JoinError to String and return if error
 - Second ?: Convert inference error and return if error
 - Only reached if both succeed!
3. `move ||`: Closure that takes ownership
 - `move` transfers ownership into the closure
 - Necessary because closure runs on different thread

MODIFIED FILE: `src-tauri/src/state.rs`

We added **one field** to the global state:

```

pub struct AudioState {
    // ... existing fields ...

    // NEW! The Gemma LLM engine (optional, loaded on demand)
    pub llm: Arc<Mutex<Option<crate::llm::LLMEngine>>,
}
  
```

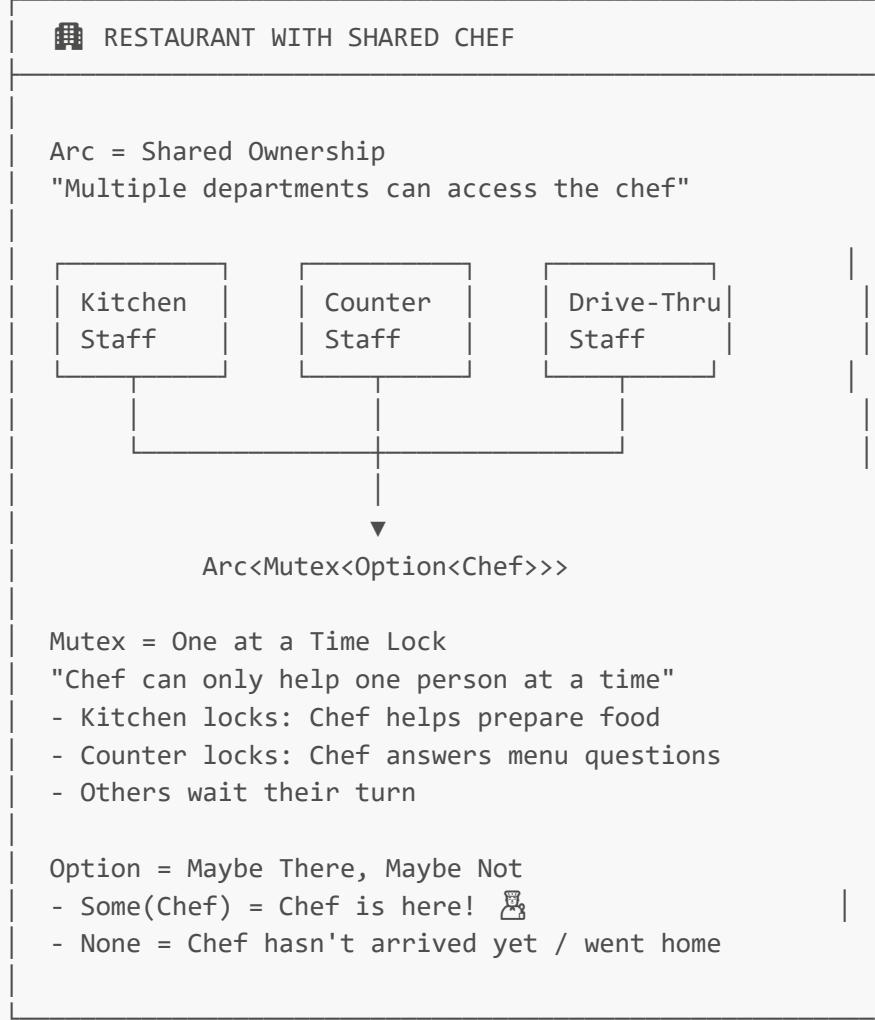
Breaking It Down: The Type Onion

```

Arc<Mutex<Option<LLMEngine>>
  |   |   |
  |   |   |   LLMEngine = The actual machine
  |   |   |
  |   |   |   Option<...> = Might be loaded, might not
  
```

- Some(engine) or None
- Mutex<...> = Thread-safe lock
 - Only one thread at a time can access
- Arc<...> = Atomic Reference Counter
 - Multiple parts of app can share ownership
 - Last one to drop cleans up

Analogy: The Shared Chef



Usage Example:

```
// Get access to LLM
let llm_guard = state.llm.lock().unwrap();
//           |           |           └ "Panic if lock poisoned"
//           |           └ "Wait and acquire lock"
//           └ MutexGuard<Option<LLMEngine>>

// Check if loaded
if llm_guard.is_some() {
```

```

    println!("LLM is ready!");
}

// Get mutable access
let mut llm_guard = state.llm.lock().unwrap();
if let Some(engine) = llm_guard.as_mut() {
    engine.run("fix this text");
}

```

MODIFIED FILE: `src-tauri/Cargo.toml`

We added **new dependencies** for machine learning:

```

# Candle ML Framework
candle-core = { git = "https://github.com/huggingface/candle.git", version =
"0.9.2" }
candle-nn = { git = "https://github.com/huggingface/candle.git", version = "0.9.2" }
candle-transformers = { git = "https://github.com/huggingface/candle.git", version =
= "0.9.2" }
tokenizers = "0.21.0"
hf-hub = "0.4"
anyhow = "1.0"

```

What are these?

Dependency	Purpose	Analogy
candle-core	Core tensor operations, GPU support	The workshop with tools
candle-nn	Neural network building blocks	Pre-made machine parts
candle-transformers	Transformer model implementations (Gemma, etc.)	Specialized machinery blueprints
tokenizers	Text ↔ Token conversion	The translator dictionary
hf-hub	Download models from Hugging Face	The model store/library
anyhow	Better error handling	The error message system

Why Candle?

MACHINE LEARNING FRAMEWORKS IN RUST

- | | |
|---------------------|-------------------|
| PyTorch (Python) | → ⚡ Candle (Rust) |
| TensorFlow (Python) | → ⚡ Candle (Rust) |

- Why Candle?
- Pure Rust (no Python needed!)
 - Fast compilation
 - Great for desktop apps
 - CUDA and CPU support
 - Quantized models (smaller, faster)
 - Made by Hugging Face (trusted ML company)

⚙️ MODIFIED FILE: `src-tauri/.cargo/config.toml`

Added compile flags to fix linking issues:

```
[target.x86_64-pc-windows-msvc]
rustflags = [
    "-L",
    "C:/Users/abdul/OneDrive/Desktop/Taurscribe/taurscribe-runtime/bin",
    "-C",
    "link-arg=/NODEFAULTLIB:LIBCMT", # ← NEW!
]

[env]
# ... existing vars ...
CFLAGS = "/MD"      # ← NEW!
CXXFLAGS = "/MD"     # ← NEW!
```

What does this do?

Problem: Windows C++ libraries conflict (LIBCMT vs MSVCRT)
 Like trying to use two different translation dictionaries
 at the same time - they contradict each other!

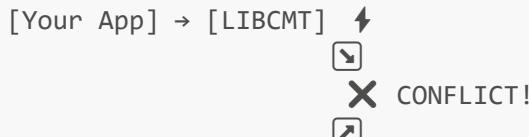
Solution: Tell the linker:

1. /NODEFAULTLIB:LIBCMT = "Don't use the default static library"
2. /MD flag = "Use the dynamic runtime library instead"

Result: Clean compilation, no linker errors!

Analogy: Choosing the Right Power Adapter

✗ Before:



[Candle] → [MSVCRT] ⚡

After:

[Your App] → [MSVCRT] ⚡



SAME SOURCE!



[Candle] → [MSVCRT] ⚡

⌚ MODIFIED FILE: `src/App.tsx` (Frontend)

Added a "**Correct Grammar**" button that appears after transcription:

```
{!isRecording && liveTranscript && (
  <div className="correction-container">
    <button
      onClick={async () => {
        setIsCorrecting(true);
        toast.loading("Correcting grammar...");
        try {
          const corrected = await invoke("correct_text", { text: liveTranscript
      });
        setLiveTranscript(corrected as string);
        toast.success("Grammar corrected!");
      } catch (e) {
        toast.error("Correction failed: " + e);
      } finally {
        setIsCorrecting(false);
      }
    }}
    disabled={isCorrecting}
    className="btn btn-correct"
    >
      {isCorrecting ? "⚡ Correcting..." : "◆ Correct Grammar"}
    </button>
  </div>
)}
```

Breaking It Down:

```
// 1. Show button only when conditions are met:
!isRecording           // Not currently recording
&& liveTranscript     // Have some transcribed text

// 2. Button click handler (async function):
onClick={async () => {
  // Show loading state
  setIsCorrecting(true);
  toast.loading("Correcting grammar...");
```

```
try {
    // Call Rust backend via Tauri
    const corrected = await invoke("correct_text", {
        text: liveTranscript
    });

    // Update UI with corrected text
    setLiveTranscript(corrected as string);
    toast.success("Grammar corrected!");

} catch (e) {
    // Handle errors gracefully
    toast.error("Correction failed: " + e);

} finally {
    // Always clean up loading state
    setIsCorrecting(false);
}
}
```

Flow Diagram:





📝 MODIFIED FILE: [src/App.css](#)

Added beautiful styling for the correction button:

```
.correction-container {  
  margin-top: 16px;  
  padding-top: 16px;  
  border-top: 1px solid var(--border-color);  
  display: flex;  
  justify-content: flex-end;  
}  
  
.btn-correct {  
  background: linear-gradient(135deg, #7c3aed 0%, #a78bfa 100%);  
  color: white;  
  box-shadow: 0 4px 15px rgba(124, 58, 237, 0.4);  
  min-width: 180px;  
  font-size: 0.9rem;  
  padding: 10px 20px;  
}  
  
.btn-correct:not(:disabled):hover {  
  box-shadow: 0 6px 20px rgba(124, 58, 237, 0.5);  
  background: linear-gradient(135deg, #8b5cf6 0%, #c4b5fd 100%);  
}
```

Visual Preview:

 Transcript

"i went too the stor yesterday"

◆ Correct Grammar
(Purple gradient ❤)

 How It All Works Together: The Complete PictureTAURSCRIBE ECOSYSTEM
With Grammar Correction

PHASE 1: RECORDING & TRANSCRIPTION

1. User speaks → Microphone captures audio
2. Audio sent to Whisper/Parakeet ASR engine
3. ASR converts speech to text
4. Raw transcript displayed: "i went too the stor yesterday"



PHASE 2: GRAMMAR CORRECTION (NEW!)

5. User clicks "◆ Correct Grammar"
6. Frontend invokes: `correct_text({ text: '...' })`
7. Request travels through Tauri IPC bridge
8. Rust backend receives request
9. Locks LLM mutex, gets engine
10. Formats prompt: "`<bos>text\n...\\ncorrected\\n`"
11. Tokenizer converts text → token IDs
12. Creates tensor on GPU/CPU
13. Runs Gemma model forward pass (1B parameters!)
14. Samples next token using temperature + top-p
15. Decodes token ID → text
16. Returns corrected text to frontend
17. UI updates: "I went to the store yesterday"

🎓 Key Learning Points for Rust Beginners

1 The Type System is Your Friend

Rust's complex types might look scary, but they prevent bugs!

```
Arc<Mutex<Option<LLMEngine>>>
```

Each layer has a purpose:

- `Option` = Handles "might not exist"
- `Mutex` = Prevents data races
- `Arc` = Enables safe sharing

2 Ownership & Borrowing

```
// Borrowing (read-only)
fn read_text(text: &str) { ... }

// Mutable borrowing (can modify)
fn fix_text(text: &mut String) { ... }

// Taking ownership (transfer)
fn consume_text(text: String) { ... }
```

3 Error Handling with Result

```
fn might_fail() -> Result<String, String> {
    if everything_ok {
        Ok("Success!".to_string())
    } else {
        Err("Something went wrong".to_string())
    }
}

// Using it:
match might_fail() {
    Ok(value) => println!("Got: {}", value),
    Err(e) => eprintln!("Error: {}", e),
}
```

```
// Or use ? for quick propagation:
let value = might_fail()?;
// Returns error automatically if failed
```

4 Async/Await for Non-Blocking Code

```
async fn load_heavy_data() -> Result<Data> {
    // This might take seconds, but won't block other tasks!
    let data = download_from_internet().await?;
    Ok(data)
}
```

5 Lifetimes (The ' Thing)

```
State<'_, AudioState>
//      └ "This reference lives for some duration"
//          Compiler figures it out automatically!"
```

Performance Characteristics

Metric	Value	Notes
Model Size	~1 GB (quantized)	Q4_K_M quantization
Load Time	2-3 seconds	First initialization
Inference Time (GPU)	50-200ms	Depends on input length
Inference Time (CPU)	500-2000ms	Much slower without GPU
Memory Usage	~2 GB RAM	Includes model + runtime
Thread Safety	<input checked="" type="checkbox"/> Yes	Mutex-protected
Concurrent Users	1 at a time	Mutex serializes access

Future Improvements & Possibilities

1. Multi-Turn Generation

Currently generates only one token. Could loop to generate full sentences:

```
pub fn run(&mut self, prompt: &str) -> Result<String> {
    let mut tokens = vec![...]; // Initial tokens

    for _ in 0..MAX_TOKENS {
        let logits = self.model.forward(&input, position)?;
```

```
        let next_token = self.logits_processor.sample(&last_logits)?;

        if next_token == EOS_TOKEN { break; } // Stop at end

        tokens.push(next_token);
    }

    self.tokenizer.decode(&tokens, true)
}
```

2. Streaming Output

Show words as they generate (like ChatGPT):

```
// Send tokens one-by-one to frontend
app_handle.emit_all("llm_token", TokenEvent { token: word })?;
```

3. Model Swapping

Let users choose different models:

- Small (135M params) - Fast
- Medium (360M params) - Balanced
- Large (1.7B params) - Best quality

4. Context Awareness

Use previous transcripts for better corrections:

```
let prompt = format!(
    "Previous: {}\nCurrent: {}\nCorrected:",
    previous_context,
    current_text
);
```

5. Batch Processing

Correct multiple transcripts at once for efficiency.

↳ Common Issues & Solutions

Issue 1: "Model file not found"

```
Error: Model file not found at: c:\...\GRMR-V3-G1B-Q4_K_M.gguf
```

Solution: Download the model and place it in the correct directory:

```
tauruscribe-runtime/
  models/
    GRMR-V3-G1B-GGUF/
      GRMR-V3-G1B-Q4_K_M.gguf
      tokenizer.json
```

Issue 2: "CUDA not available"

```
[LLM] Using device: Cpu
```

This is OK! The app automatically falls back to CPU. It will be slower but still work.

To enable GPU:

1. Install CUDA Toolkit (NVIDIA only)
2. Rebuild the app
3. Check with `nvidia-smi` that your GPU is recognized

Issue 3: "Linking error: LIBCMT conflict"

```
error: duplicate symbol found in LIBCMT and MSVCRT
```

Solution: Already fixed in `.cargo/config.toml`!

```
rustflags = ["-C", "link-arg=/NODEFAULTLIB:LIBCMT"]
```

Issue 4: Inference is very slow

Possible causes:

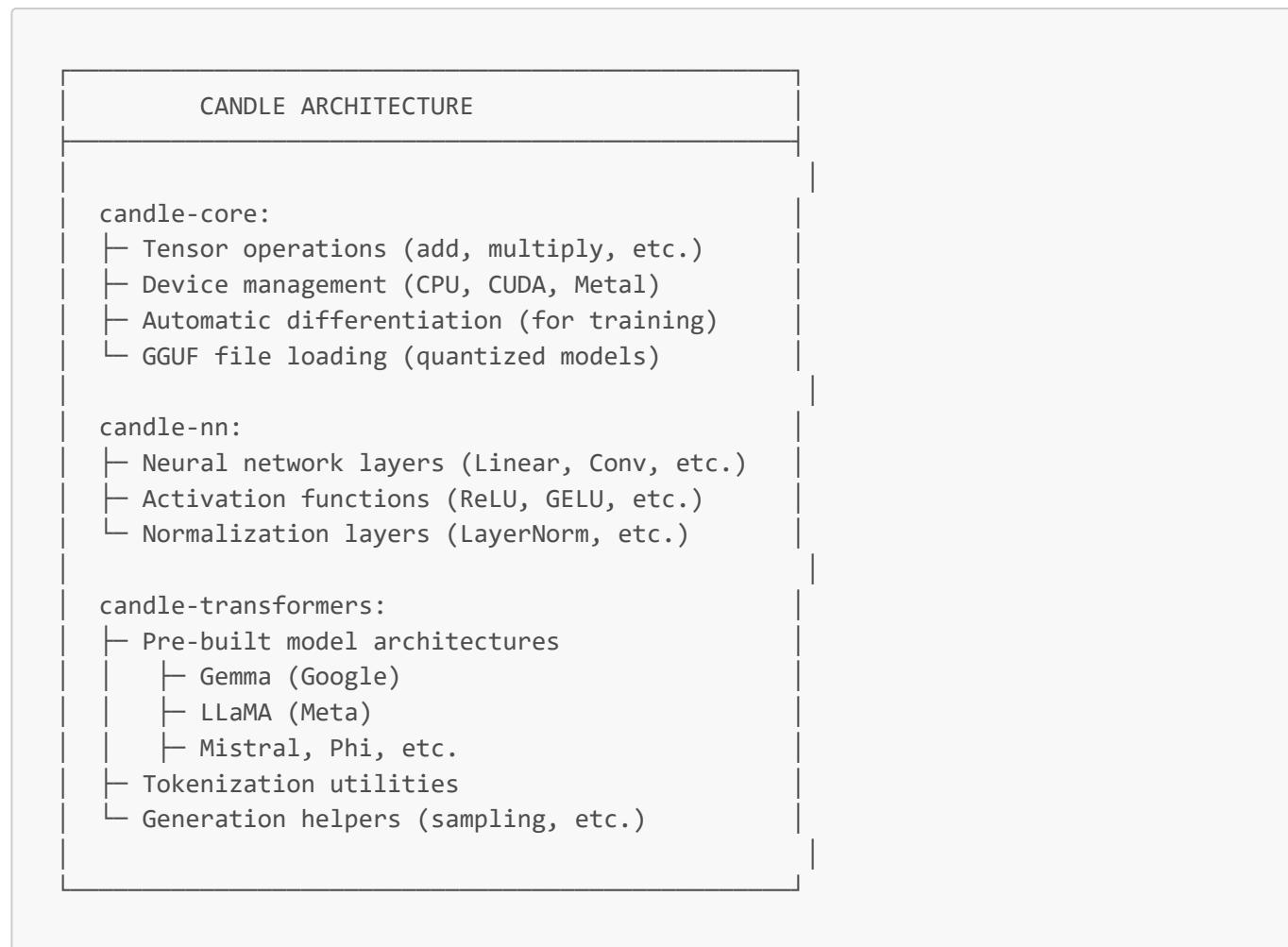
1. Using CPU instead of GPU (50x slower)
2. Model is too large
3. Input text is very long

Solutions:

- Check if GPU is enabled: Look for "Using device: Cuda(0)"
- Try a smaller model (Q4_K_M is already quantized)
- Limit input text length to 512 tokens

Dependencies Deep Dive

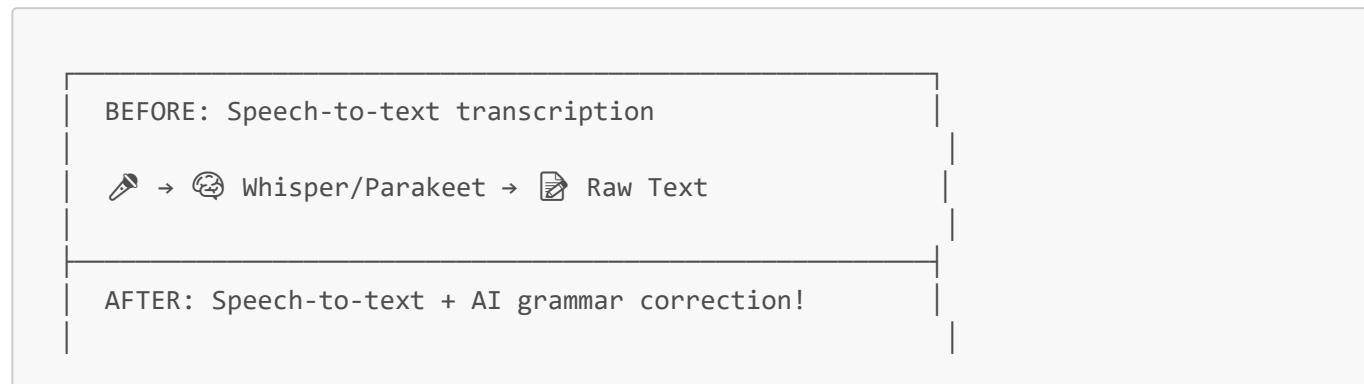
Candle Framework

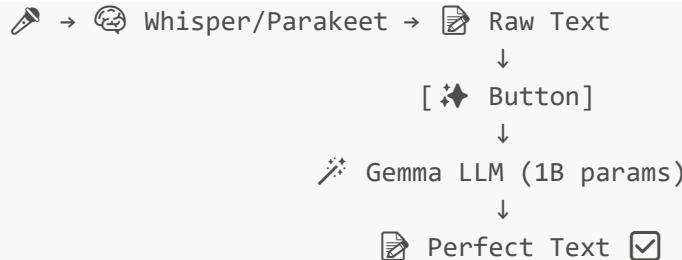


Key External Libraries

Library	Purpose	Used For
<code>tokenizers</code>	Hugging Face tokenizer library	Convert text ↔ token IDs
<code>hf-hub</code>	Download models from Hugging Face	Model distribution
<code>anyhow</code>	Ergonomic error handling	Better error messages

⌚ Summary: What We Added





Files Created:

- `src-tauri/src/llm.rs` (LLM engine)
- `src-tauri/src/commands/llm.rs` (Command handlers)

Files Modified:

- `src-tauri/src/state.rs` (Added LLM to global state)
- `src-tauri/src/lib.rs` (Registered commands)
- `src-tauri/Cargo.toml` (Added Candle dependencies)
- `src-tauri/.cargo/config.toml` (Fixed linking)
- `src/App.tsx` (Added correction button)
- `src/App.css` (Styled the button)

Total Lines Added: ~2,700 lines

- Rust code: ~200 lines
- Dependencies (Cargo.lock): ~800 lines
- Documentation (ARCHITECTURE.md): ~1,670 lines
- Frontend: ~30 lines

🎓 Recommended Learning Path

If you're new to Rust and want to understand this code better:

1. Learn Rust Basics (1-2 weeks)

- Variables and types
- Functions and ownership
- Structs and enums
- Error handling with Result

2. Learn Async Rust (1 week)

- async/await syntax
- Tokio/async-runtime basics
- spawn_blocking vs spawn

3. Learn Concurrency (1 week)

- Arc and Mutex

- Thread safety
- Channels

4. Machine Learning Basics (Optional)

- What are tensors?
- How do neural networks work?
- What is quantization?

Resources:

- [The Rust Book](#)
 - [Async Book](#)
 - [Candle Examples](#)
-

Congratulations!

You now understand how Taurscribe added AI-powered grammar correction! You learned about:

- Rust async programming
- Machine learning inference
- Thread-safe state management
- Frontend-backend communication
- Error handling patterns
- The Candle ML framework
- Neural network basics

Keep exploring and building amazing things!

Complete Rust File Reference (Detailed)

This section provides **in-depth, line-by-line documentation** for every Rust file in the Taurscribe codebase. Each file is explained with:

- Complete struct/enum definitions with field explanations
 - Every function with step-by-step breakdowns
 - Where types and values come from (imports vs local definitions)
 - Visual flow diagrams
 - Rust concepts explained
-

File Structure Overview

```
src-tauri/
├── build.rs                  # Build-time configuration (CUDA paths, macOS
targets)
└── src/
    ├── main.rs                # Entry point - calls lib::run()
    └── lib.rs                 # App orchestrator - initializes everything
```

```

    ├── state.rs          # Global application state (AudioState)
    ├── types.rs          # Shared type definitions (enums, structs)
    └── utils.rs          # Helper functions (clean_transcript,
get_recordings_dir)
        ├── audio.rs       # Audio stream wrappers (SendStream, RecordingHandle)
        ├── whisper.rs     # Whisper AI engine manager (781 lines)
        ├── parakeet.rs    # Parakeet/Nemotron ASR manager (615 lines)
        ├── vad.rs          # Voice Activity Detection (215 lines)
        ├── llm.rs          # Gemma LLM for grammar correction (98 lines)
        └── commands/
            ├── mod.rs       # Module re-exports
            ├── recording.rs # start_recording, stop_recording commands
            ├── transcription.rs # list_sample_files, benchmark_test commands
            ├── models.rs      # list_models, switch_model commands
            ├── settings.rs   # get_backend_info, set_active_engine commands
            ├── llm.rs          # init_llm, correct_text commands
            └── misc.rs         # greet (test command)
        └── hotkeys/
            ├── mod.rs       # Module re-exports
            └── listener.rs  # Global hotkey listener (Ctrl+Win)
    └── tray/
        ├── mod.rs       # Module re-exports
        └── icons.rs     # System tray setup and icon management

```

src-tauri/src/main.rs (7 lines)

Purpose: The absolute entry point of the Rust application.

Complete File

```

// Line 1-2: Prevent Windows from opening a console window in release builds
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]

// Line 4-6: Main function - calls the library's run() function
fn main() {
    taurscribe_lib::run()
}

```

Line-by-Line Explanation

Line 1-2: Compiler Attribute

```
#![cfg_attr(not(debug_assertions), windows_subsystem = "windows")]
```

Component	Meaning
-----------	---------

Component	Meaning
<code>#[]</code>	Inner attribute - applies to entire file/crate
<code>cfg_attr(condition, attribute)</code>	Conditionally apply an attribute
<code>not(debug_assertions)</code>	True when building in release mode
<code>windows_subsystem = "windows"</code>	Don't show console window on Windows

Why: In release mode, the app runs as a GUI-only application without a console window.

Lines 4-6: Main Function

```
fn main() {
    taurscribe_lib::run()
}
```

Component	Meaning
<code>fn main()</code>	Standard Rust entry point
<code>taurscribe_lib</code>	The library crate name (from <code>Cargo.toml</code> → <code>name = "taurscribe_lib"</code>)
<code>::run()</code>	Calls the <code>run()</code> function exported from <code>lib.rs</code>

Flow Diagram



src-tauri/src/lib.rs (119 lines)

Purpose: The main orchestrator that initializes all AI engines and starts the Tauri application.

Imports & Module Declarations (Lines 1-18)

```
// Lines 1-12: Declare submodules
mod audio;          // Audio stream handling
mod commands;       // Tauri command handlers
mod hotkeys;        // Global hotkey listener
mod llm;            // Gemma LLM engine
mod parakeet;       // Parakeet ASR engine
mod state;          // Global state struct
mod tray;           // System tray management
mod types;          // Shared type definitions
mod utils;          // Helper functions
mod vad;             // Voice Activity Detection
mod whisper;         // Whisper AI engine

// Lines 14-18: Import specific items from our modules
use parakeet::ParakeetManager;
use state::AudioState;
use vad::VADManager;
use whisper::WhisperManager;
```

The `run()` Function (Lines 20-118)

This is the main entry point called from `main.rs`.

Step 1: Initialize Whisper AI (Lines 24-47)

```
// Line 24-26: Create a new WhisperManager (not loaded yet)
println!("[INFO] Initializing Whisper transcription engine..."); 
let whisper = WhisperManager::new();

// Lines 28-37: Load model in a separate thread with large stack
let (whisper, init_result) = std::thread::Builder::new()
    .stack_size(8 * 1024 * 1024) // 8 MiB stack (needed for large models)
    .spawn(move || {
        let mut whisper = whisper;           // Take ownership
        let res = whisper.initialize(None); // Load default model
        (whisper, res)                   // Return both
    })
    .expect("Failed to spawn whisper init thread")
    .join()                           // Wait for thread to finish
    .expect("Whisper init thread panicked");

// Lines 39-47: Handle initialization result
match init_result {
    Ok(backend_msg) => {
        println!("[SUCCESS] {}", backend_msg); // e.g., "Backend: CUDA"
    }
    Err(e) => {
        eprintln!("[ERROR] Failed to initialize Whisper: {}", e);
    }
}
```

```

        eprintln!("Transcription will be disabled.");
    }
}

```

Why separate thread?: Loading AI models requires large stack space (8MB vs default 2MB) to avoid stack overflow.

Step 2: Initialize VAD (Lines 49-55)

```

println!("[INFO] Initializing Voice Activity Detection...");
let vad = VADManager::new().unwrap_or_else(|e| {
    eprintln!("[ERROR] Failed to initialize VAD: {}", e);
    panic!("VAD initialization failed"); // VAD is required
});
println!("[SUCCESS] VAD initialized successfully");

```

Step 3: Initialize Parakeet (Lines 57-68)

```

println!("[INFO] Initializing Parakeet ASR manager...");
let mut parakeet = ParakeetManager::new();

// Try to load Nemotron model first, then fallback to any available
println!("[INFO] Attempting to auto-load Parakeet model...");
match parakeet.initialize(Some("nemotron:nemotron")) {
    Ok(msg) => println!("[SUCCESS] {}", msg),
    Err(_) => match parakeet.initialize(None) { // Fallback
        Ok(msg) => println!("[SUCCESS] Fallback load: {}", msg),
        Err(e) => eprintln!("[WARN] No Parakeet models loaded: {}", e),
    },
}

```

Step 4: Build Tauri App (Lines 70-118)

```

tauri::Builder::default()
    // Register plugins
    .plugin(tauri_plugin_opener::init()) // Open URLs/files
    .plugin(tauri_plugin_store::Builder::default().build()) // Settings storage

    // Register global state
    .manage(AudioState::new(whisper, parakeet, vad))

    // Setup callback (runs once after window is created)
    .setup(|app| {
        tray::setup_tray(app)?; // Create system tray

        // Start hotkey listener in background
    })
}

```

```

let app_handle = app.handle().clone();
std::thread::spawn(move || {
    hotkeys::start_hotkey_listener(app_handle);
});

println!("[INFO] Global hotkey listener started (Ctrl+Win to record)");
Ok(())
})

// Window close handler (minimize to tray instead)
.on_window_event(|window, event| {
    if let tauri::WindowEvent::CloseRequested { api, .. } = event {
        let _ = window.hide(); // Hide window
        api.prevent_close(); // Don't actually close
        println!("[INFO] Window minimized to tray");
    }
})

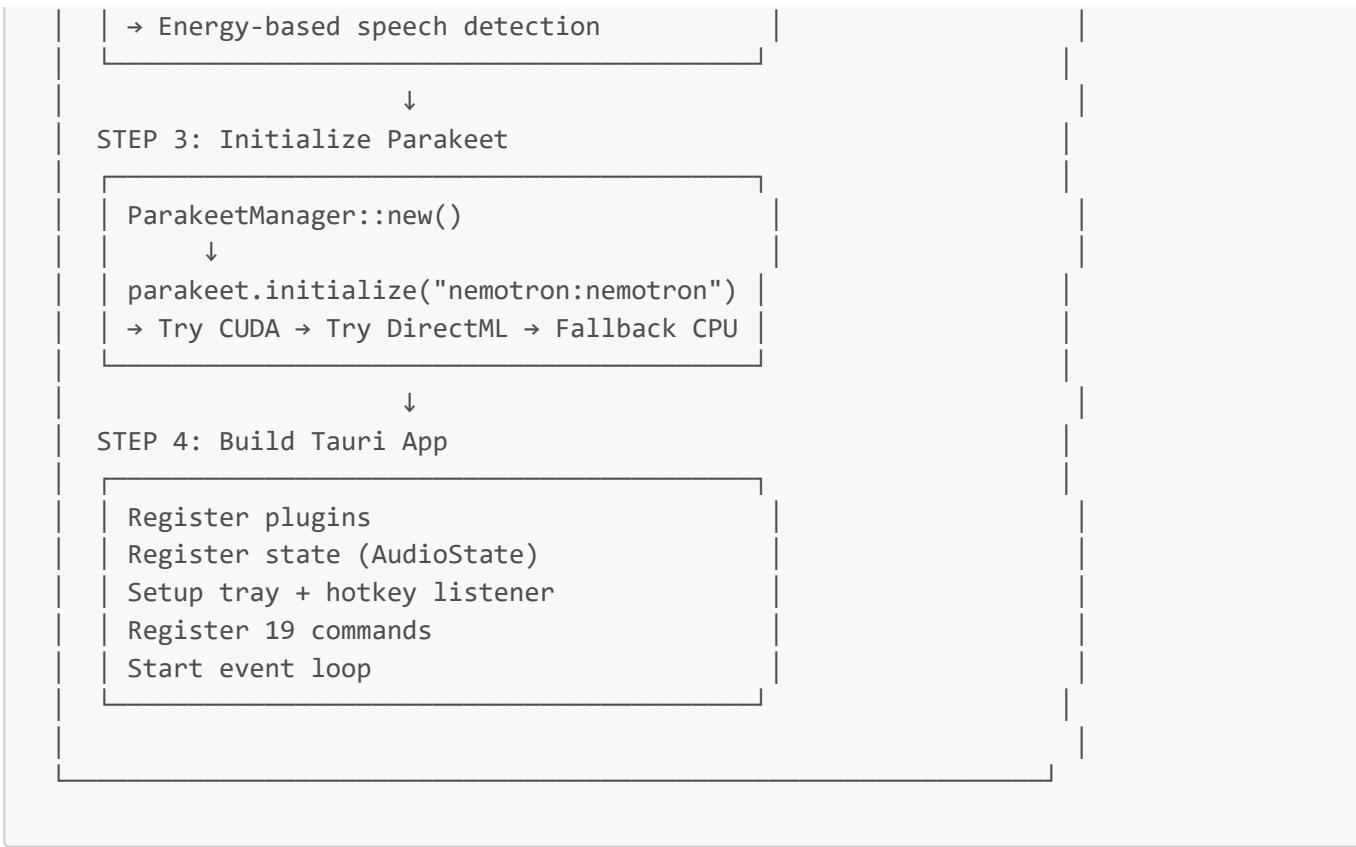
// Register all Tauri commands (callable from frontend)
.invoke_handler(tauri::generate_handler![
    commands::greet,
    commands::start_recording,
    commands::stop_recording,
    // ... 15 more commands
])
}

// Run the app!
.run(tauri::generate_context!())
.expect("error while running tauri application");

```

Complete Flow Diagram





src-tauri/src/state.rs (55 lines)

Purpose: Defines the global application state that persists throughout the app's lifetime.

Imports (Lines 1-6)

```

use crate::audio::RecordingHandle;           // From audio.rs
use crate::parakeet::ParakeetManager;         // From parakeet.rs
use crate::types::{ASREngine, AppState};      // From types.rs
use crate::vad::VADManager;                  // From vad.rs
use crate::whisper::WhisperManager;          // From whisper.rs
use std::sync::{Arc, Mutex};                 // Thread-safe wrappers
  
```

The **AudioState** Struct (Lines 8-38)

```

pub struct AudioState {
    // Field 1: Active recording (None if not recording)
    pub recording_handle: Mutex<Option<RecordingHandle>>,

    // Field 2: Whisper AI engine (shared across threads)
    pub whisper: Arc<Mutex<WhisperManager>>,

    // Field 3: Parakeet AI engine (shared across threads)
    pub parakeet: Arc<Mutex<ParakeetManager>>,

    // Field 4: Voice Activity Detector (shared)
  
```

```

pub vad: Arc<Mutex<VADManager>>,

// Field 5: Path to last recorded WAV file
pub last_recording_path: Mutex<Option<String>>,

// Field 6: Current app state (Ready/Recording/Processing)
pub current_app_state: Mutex<AppState>,

// Field 7: Which ASR engine is active (Whisper or Parakeet)
pub active_engine: Mutex<ASREngine>,

// Field 8: Accumulated transcript during recording
pub session_transcript: Arc<Mutex<String>>,

// Field 9: Optional LLM engine (loaded on demand)
pub llm: Arc<Mutex<Option<crate::llm::LLMEngine>>>,
}

```

Field Details Table

Field	Type	Initial Value	Purpose
recording_handle	Mutex<Option<RecordingHandle>>	None	Holds mic stream when recording
whisper	Arc<Mutex<WhisperManager>>	Initialized	Whisper AI engine
parakeet	Arc<Mutex<ParakeetManager>>	Initialized	Parakeet ASR engine
vad	Arc<Mutex<VADManager>>	Initialized	Voice Activity Detection
last_recording_path	Mutex<Option<String>>	None	Path to saved WAV file
current_app_state	Mutex<AppState>	Ready	Tray icon state
active_engine	Mutex<ASREngine>	Whisper	Selected transcription engine
session_transcript	Arc<Mutex<String>>	Empty string	Live transcript accumulator
llm	Arc<Mutex<Option<LLMEngine>>>	None	Gemma LLM (lazy loaded)

Why `Arc<Mutex<T>>`?

`Arc<Mutex<T>>` Explained

Arc (Atomic Reference Counting):

- |- Allows MULTIPLE owners of the same data
- |- Reference count is thread-safe (atomic operations)

```

    └ Data is deallocated when count reaches 0

Mutex (Mutual Exclusion):
├ Only ONE thread can access data at a time
└ .lock() waits if another thread has the lock
└ .unwrap() panics if lock is poisoned (rare)

Combined:
├ Multiple threads can SHARE ownership (Arc)
└ But only ONE can ACCESS at a time (Mutex)

```

Constructor (Lines 40-54)

```

impl AudioState {
    pub fn new(whisper: WhisperManager, parakeet: ParakeetManager, vad: VADManager) -> Self {
        Self {
            recording_handle: Mutex::new(None),
            whisper: Arc::new(Mutex::new(whisper)),
            parakeet: Arc::new(Mutex::new(parakeet)),
            vad: Arc::new(Mutex::new(vad)),
            last_recording_path: Mutex::new(None),
            current_app_state: Mutex::new(AppState::Ready),
            active_engine: Mutex::new(ASREngine::Whisper),
            session_transcript: Arc::new(Mutex::new(String::new())),
            llm: Arc::new(Mutex::new(None)),
        }
    }
}

```

src-tauri/src/types.rs (31 lines)

Purpose: Shared type definitions used across multiple files.

Complete File with Explanations

```

/// AppState - Represents the current state of the application
/// Used to determine tray icon color
#[derive(Debug, Clone, Copy, PartialEq)] // Derive common traits
pub enum AppState {
    Ready, // ⚪ Green - Waiting for user input
    Recording, // ⚫ Red - Microphone is active
    Processing, // ⚩ Yellow - Transcribing audio
}

/// ASREngine - The two speech recognition engines available

```

```

#[derive(Debug, Clone, Copy, serde::Serialize, serde::Deserialize, PartialEq)]
pub enum ASREngine {
    Whisper, // OpenAI Whisper (via whisper.cpp)
    Parakeet, // NVIDIA Parakeet (via onnxruntime)
}

/// TranscriptionChunk - Payload sent to frontend during live transcription
#[derive(Debug, Clone, serde::Serialize)]
pub struct TranscriptionChunk {
    pub text: String, // The transcribed text
    pub processing_time_ms: u32, // How long it took (milliseconds)
    pub method: String, // "Whisper" or "Parakeet"
}

/// SampleFile - Information about a test audio file
#[derive(serde::Serialize)]
pub struct SampleFile {
    pub name: String, // Filename, e.g., "test.wav"
    pub path: String, // Full path on disk
}

```

Derive Macros Explained

Macro	Purpose
Debug	Allows <code>println!("{}: {}")</code> , <code>value</code>
Clone	Allows <code>.clone()</code> to copy the value
Copy	Allows implicit copying (small types only)
PartialEq	Allows <code>==</code> and <code>!=</code> comparisons
serde::Serialize	Allows conversion to JSON (for frontend)
serde::Deserialize	Allows conversion from JSON

src-tauri/src/utils.rs (47 lines)

Purpose: Helper utility functions used across the application.

Function 1: `clean_transcript()` (Lines 1-31)

Purpose: Post-processes raw ASR output to fix common formatting issues.

```

pub fn clean_transcript(text: &str) -> String {
    let mut cleaned = text.trim().to_string();

    // Fix 1: Remove space before punctuation
    cleaned = cleaned.replace(" ,", ","); // "hello , world" → "hello, world"
    cleaned = cleaned.replace(" .", "."); // "hello ." → "hello."
    cleaned = cleaned.replace(" ?", "?");
}

```

```

cleaned = cleaned.replace(" !", "!");
cleaned = cleaned.replace(" %", "%"); // "50 %" → "50%"

// Fix 2: Remove double spaces
while cleaned.contains("  ") {
    cleaned = cleaned.replace("  ", " ");
}

// Fix 3: Capitalize first letter
if let Some(first) = cleaned.chars().next() {
    if first.is_lowercase() {
        let mut c = cleaned.chars();
        cleaned = match c.next() {
            None => String::new(),
            Some(f) => f.to_uppercase().collect::<String>() + c.as_str(),
        };
    }
}

cleaned
}

```

Function 2: `get_recordings_dir()` (Lines 33-46)

Purpose: Returns the path to save recordings, creating it if needed.

```

pub fn get_recordings_dir() -> Result<std::path::PathBuf, String> {
    // Get OS-specific AppData folder
    // Windows: C:\Users\Name\AppData\Local
    // macOS: ~/Library/Application Support
    // Linux: ~/.local/share
    let app_data = dirs::data_local_dir()
        .ok_or("Could not find AppData directory")?;

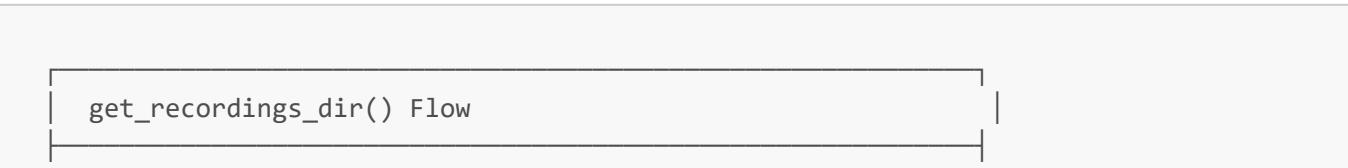
    // Create: AppData/Taurscribe/temp
    let recordings_dir = app_data.join("Taurscribe").join("temp");

    // Create folder if it doesn't exist
    std::fs::create_dir_all(&recordings_dir)
        .map_err(|e| format!("Failed to create recordings directory: {}", e))?;

    Ok(recordings_dir)
}

```

Flow Diagram



```

graph TD
    A["get_recordings_dir() Flow"]

```

```

Windows:
C:\Users\abdul\AppData\Local
    ↳ Taurscribe
        ↳ temp
            ↳ recording_1706850000.wav

macOS:
~/Library/Application Support
    ↳ Taurscribe/temp/

```

src-tauri/src/audio.rs (17 lines)

Purpose: Wrapper types for thread-safe audio stream handling.

Complete File

```

use crossbeam_channel::Sender;

/// SendStream - Wrapper to make cpal::Stream thread-safe
/// By default, audio streams can't be moved between threads
/// We implement Send + Sync manually to enable this
#[allow(dead_code)]
pub struct SendStream(pub cpal::Stream);

// UNSAFE: We promise to only drop this, never access across threads
unsafe impl Send for SendStream {} // Can be moved to another thread
unsafe impl Sync for SendStream {} // Can be shared between threads

/// RecordingHandle - Holds all resources needed during active recording
pub struct RecordingHandle {
    pub stream: SendStream,           // The mic connection (keeps recording
alive)
    pub file_tx: Sender<Vec<f32>>, // Channel to file writer thread
    pub whisper_tx: Sender<Vec<f32>>, // Channel to AI transcription thread
}

```

Why unsafe impl Send/Sync?

Thread Safety in Rust

By default, `cpal::Stream` is NOT thread-safe because:

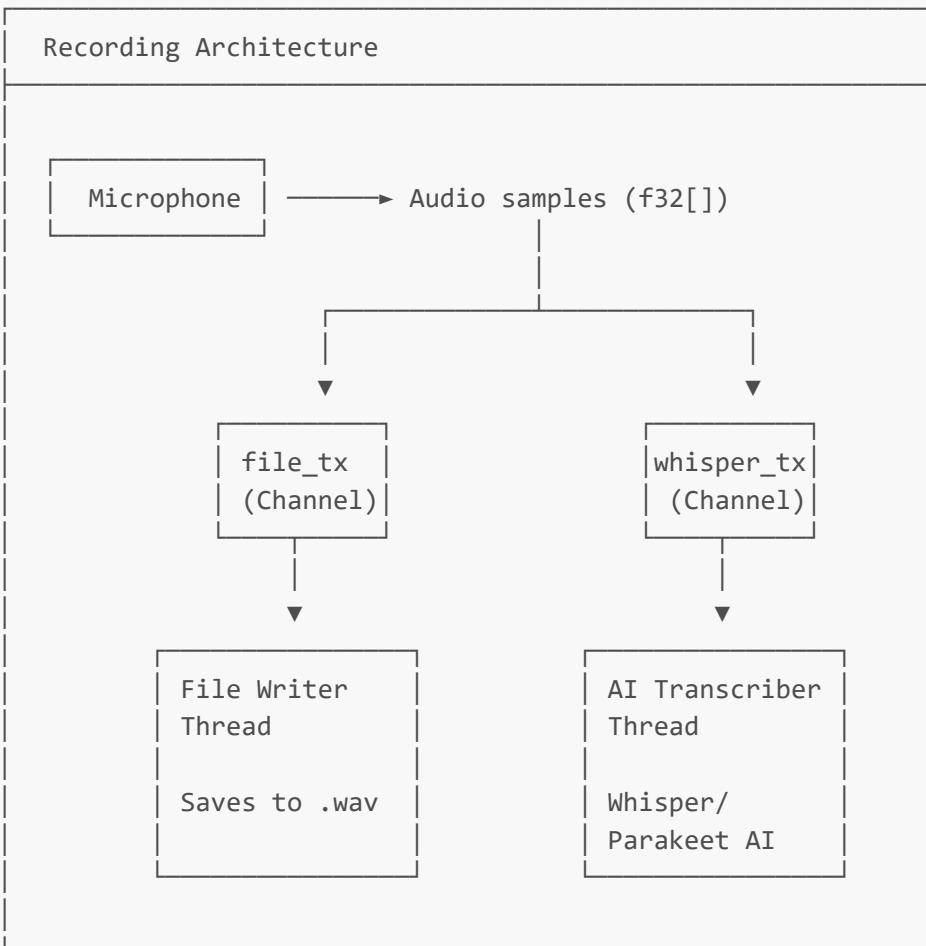
- It contains raw pointers to audio hardware
- The compiler can't verify safety automatically

We manually implement `Send + Sync` because:

- We ONLY drop the stream (never read/write)
- Dropping is always safe from any thread
- We control all access to this wrapper

⚠ WARNING: This is unsafe! Only do this if you're 100% sure about the safety guarantees.

How RecordingHandle Works



◇ `src-tauri/src/whisper.rs` (781 lines)

Purpose: Complete Whisper AI transcription engine manager.

Imports (Lines 1-8)

```

use rubato::{
    Resampler, SincFixedIn, SincInterpolationParameters,
    SincInterpolationType, WindowFunction,
}; // Audio resampling library
use std::ffi::c_void; // C interop (raw pointers)
use std::os::raw::c_char; // C interop (strings)
use whisper_rs::{
    set_log_callback, FullParams, SamplingStrategy,
    WhisperContext, WhisperContextParameters,
}; // Whisper AI library bindings

```

Enum: [GpuBackend](#) (Lines 13-31)

```

#[derive(Debug, Clone)]
pub enum GpuBackend {
    Cuda, // NVIDIA GPUs - fastest
    Vulkan, // AMD/Intel/Other GPUs
    Cpu, // Processor - slowest fallback
}

impl std::fmt::Display for GpuBackend {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            GpuBackend::Cuda => write!(f, "CUDA"),
            GpuBackend::Vulkan => write!(f, "Vulkan"),
            GpuBackend::Cpu => write!(f, "CPU"),
        }
    }
}

```

Struct: [ModelInfo](#) (Lines 33-41)

```

#[derive(Debug, Clone, serde::Serialize)]
pub struct ModelInfo {
    pub id: String, // "tiny.en-q5_1"
    pub display_name: String, // "Tiny English (Q5_1)"
    pub file_name: String, // "ggml-tiny.en-q5_1.bin"
    pub size_mb: f32, // 75.5
}

```

Struct: [WhisperManager](#) (Lines 43-50)

```

pub struct WhisperManager {
    context: Option<WhisperContext>, // AI model (None if not loaded)
    last_transcript: String, // Accumulated context for coherence
    backend: GpuBackend, // Current hardware (CPU/CUDA/Vulkan)
}

```

```

    current_model: Option<String>,      // "tiny.en-q5_1"
    resampler: Option<(u32, usize, Box<SincFixedIn<f32>>)>, // Cached resampler
}

```

Field Lifecycle

WhisperManager Field Lifecycle			
Field	new()	initialize()	transcribe()
context	None	Some(ctx)	Read
last_transcript	""	(unchanged)	Appended
backend	Cpu	Cuda/Vulkan/Cpu	Read
current_model	None	Some("model")	Read
resampler	None	(unchanged)	Created*

* Resampler is lazily created on first transcribe_chunk call

Method: new() (Lines 65-75)

```

pub fn new() -> Self {
    Self {
        context: None,           // No AI loaded
        last_transcript: String::new(), // Empty context
        backend: GpuBackend::Cpu, // Assume CPU
        current_model: None,     // No model
        resampler: None,         // Create when needed
    }
}

```

Method: get_models_dir() (Lines 77-108)

Purpose: Finds the directory containing Whisper model files.

```

fn get_models_dir() -> Result<std::path::PathBuf, String> {
    // Check these paths in order
    let possible_paths = [
        "taurscribe-runtime/models", // From project root
        "../taurscribe-runtime/models", // One level up
        "../../taurscribe-runtime/models", // Two levels up
    ];
    for path in possible_paths {

```

```

// Try to convert to absolute path
if let Ok(canonical) = std::fs::canonicalize(path) {
    if canonical.is_dir() {
        // Verify it's the right folder
        if let Ok(entries) = std::fs::read_dir(&canonical) {
            for entry in entries.flatten() {
                if let Some(name) = entry.file_name().to_str() {
                    // Look for ggml-*.bin files
                    if name.starts_with("ggml-") && name.ends_with(".bin")
{
                }
}
}
}
}
}

Err("Could not find models directory containing ggml models".to_string())
}

```

Method: `initialize()` (Lines 225-277)

Purpose: Load a Whisper model from disk into memory (and GPU).

```

pub fn initialize(&mut self, model_id: Option<&str>) -> Result<String, String> {
    // Step 1: Silence C++ logs
    unsafe { set_log_callback(None, std::ptr::null_mut()); }

    // Step 2: Find models directory
    let models_dir = Self::get_models_dir()?;

    // Step 3: Determine which model to load
    let target_model = model_id.unwrap_or("tiny.en-q5_1");
    let file_name = format!("ggml-{}.bin", target_model);
    let absolute_path = models_dir.join(&file_name);

    // Step 4: Verify file exists
    if !absolute_path.exists() {
        return Err(format!("Model file not found: {}", absolute_path.display()));
    }

    // Step 5: Try GPU first, fallback to CPU
    let (ctx, backend) = self
        .try_gpu(&absolute_path)
        .or_else(|_| self.try_cpu(&absolute_path))?;

    // Step 6: Save loaded state
    self.context = Some(ctx);
    self.backend = backend.clone();
    self.current_model = Some(target_model.to_string());
}

```

```

// Step 7: GPU warm-up (first inference is slow)
let warmup_audio = vec![0.0_f32; 16000]; // 1 second of silence
self.transcribe_chunk(&warmup_audio, 16000).ok();

Ok(format!("Backend: {}", backend))
}

```

Method: `transcribe_chunk()` (Lines 343-458)

Purpose: Real-time transcription of audio chunks (6 seconds).

```

pub fn transcribe_chunk(
    &mut self,
    samples: &[f32],           // Audio data
    input_sample_rate: u32,   // e.g., 48000 Hz
) -> Result<String, String> {
    // Get loaded model context
    let ctx = self.context.as_mut()
        .ok_or("Whisper context not initialized")?;

    // STEP 1: Resample to 16kHz if needed
    let audio_data = if input_sample_rate != 16000 {
        // Create/reuse resampler
        // ... resampling logic ...
    } else {
        samples.to_vec()
    };

    // STEP 2: Create transcription state
    let mut state = ctx.create_state()?;

    // STEP 3: Configure parameters
    let mut params = FullParams::new(SamplingStrategy::Greedy { best_of: 1 });
    params.set_n_threads(4);           // CPU threads
    params.set_translate(false);       // Don't translate
    params.set_language(Some("en"));   // English
    params.set_print_special(false);   // No <SOT> tokens

    // STEP 4: Context prompting (coherence)
    if !self.last_transcript.is_empty() {
        params.set_initial_prompt(&self.last_transcript);
    }

    // STEP 5: Run inference
    state.full(params, &audio_data)?;

    // STEP 6: Extract text
    let num_segments = state.full_n_segments();
    let mut transcript = String::new();
    for i in 0..num_segments {
        if let Some(segment) = state.get_segment(i) {

```

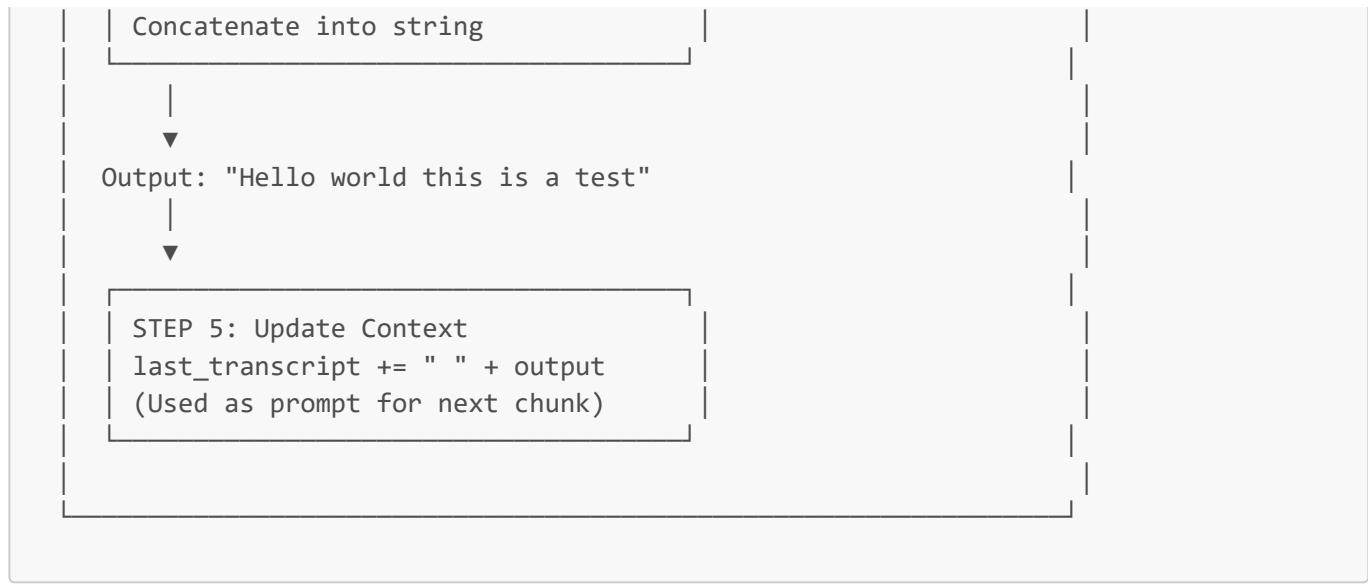
```
        transcript.push_str(&segment.to_string());
    }

// STEP 7: Update context for next chunk
let final_text = transcript.trim().to_string();
if !final_text.is_empty() {
    if !self.last_transcript.is_empty() {
        self.last_transcript.push(' ');
    }
    self.last_transcript.push_str(&final_text);
}

Ok(final_text)
}
```

Transcription Flow Diagram





src-tauri/src/parakeet.rs (615 lines)

Purpose: NVIDIA Parakeet/Nemotron ASR engine manager.

Key Differences from Whisper

Aspect	Whisper	Parakeet
Model Format	.bin (GGML)	.onnx (ONNX Runtime)
Streaming	6-second chunks	560ms chunks
GPU Backend	CUDA/Vulkan	CUDA/DirectML
Latency	~200ms	~50ms
Model Types	Single	Nemotron, CTC, EOU, TDT

Struct: **ParakeetManager** (Lines 52-57)

```

pub struct ParakeetManager {
    model: Option<LoadedModel>, // Loaded AI model
    model_name: Option<String>, // "nemotron:nemotron"
    backend: GpuBackend, // Cuda/DirectML/Cpu
    resampler: Option<(u32, usize, Box<SyncFixedIn<f32>>)>,
}
  
```

Enum: **LoadedModel** (Lines 35-40)

```

enum LoadedModel {
    Nemotron(Nemotron), // Streaming-optimized
    Ctc(Parakeet), // CTC decoder
    Eou(ParakeetEOU), // End-of-utterance detection
  
```

```

        Tdt(ParakeetTDT),           // Token-based decoding
    }

```

Method: `transcribe_chunk()` (Lines 528-613)

Purpose: Transcribe small audio chunks in real-time.

```

pub fn transcribe_chunk(
    &mut self,
    samples: &[f32],
    sample_rate: u32,
) -> Result<String, String> {
    // Step 1: Resample to 16kHz
    let audio = if sample_rate != 16000 {
        // ... resampling logic ...
    } else {
        samples.to_vec()
    };

    // Step 2: Transcribe based on model type
    if let Some(model) = &mut self.model {
        match model {
            LoadedModel::Nemotron(m) => {
                // Process in 560ms chunks (8960 samples)
                let mut transcript = String::new();
                const CHUNK_SIZE: usize = 8960;
                for chunk in audio.chunks(CHUNK_SIZE) {
                    transcript.push_str(&m.transcribe_chunk(chunk)?);
                }
                Ok(transcript)
            }
            LoadedModel::Ctc(m) => {
                // Full chunk processing
                let result = m.transcribe_samples(audio, 16000, 1, None)?;
                Ok(result.text)
            }
            // ... other model types ...
        }
    } else {
        Err("No model loaded".to_string())
    }
}

```

src-tauri/src/vad.rs (215 lines)

Purpose: Voice Activity Detection - distinguishes speech from silence.

Struct: `VADManager` (Lines 8-10)

```
pub struct VADManager {
    threshold: f32, // Volume level that counts as speech (0.005)
}
```

Method: `is_speech()` (Lines 78-98)

Purpose: Determine if an audio chunk contains speech.

```
pub fn is_speech(&mut self, audio: &[f32]) -> Result<f32, String> {
    // Calculate RMS (Root Mean Square) energy
    let sum_squares: f32 = audio.iter().map(|&x| x * x).sum();
    let rms = (sum_squares / audio.len() as f32).sqrt();

    // Convert to probability (0.0 = silence, 1.0 = speech)
    let prob = if rms < self.threshold {
        0.0 // Too quiet
    } else if rms > self.threshold * 5.0 {
        1.0 // Very loud
    } else {
        // Linear interpolation
        ((rms - self.threshold) / (self.threshold * 4.0)).min(1.0)
    };

    Ok(prob)
}
```

VAD Algorithm Visualization

Energy-Based VAD Algorithm			
Audio Signal:			
~~~~~	~~~~~	~~~~~	~~~~~
^^ speech ^^	^^ silence ^^	^^ speech ^^	^^ silent ^^
RMS Energy:			
0.05	0.001	0.04	0.002
Threshold = 0.005			
Result:			
SPEECH (1.0)	SILENCE (0.0)	SPEECH (1.0)	SILENCE (0.0)

## Method: `get_speech_timestamps()` (Lines 100-213)

**Purpose:** Find exactly when speech occurs in a full recording.

**Returns:** `Vec<(start_seconds, end_seconds)>`

**Example:** `[(0.5, 3.2), (5.0, 8.7)]` means speech from 0.5-3.2s and 5.0-8.7s

## src-tauri/src/llm.rs (98 lines)

**Purpose:** Gemma LLM engine for grammar correction.

Struct: `LLMEngine` (Lines 12-17)

```
pub struct LLMEngine {
    model: model::ModelWeights,           // Neural network weights
    tokenizer: Tokenizer,                // Text ⇌ Token IDs
    device: Device,                     // CPU or CUDA
    logits_processor: LogitsProcessor, // Sampling (temperature, top-p)
}
```

## Method: `new()` (Lines 19-63)

```
pub fn new() -> Result<Self> {
    // Hardcoded model path
    let base_path = PathBuf::from(
        r"c:\Users\abdul\OneDrive\Desktop\Taurscribe\taurscribe-
runtime\models\GRMR-V3-G1B-GGUF",
    );
    let model_path = base_path.join("GRMR-V3-G1B-Q4_K_M.gguf");
    let tokenizer_path = base_path.join("tokenizer.json");

    // Select device (try CUDA first)
    let device = Device::new_cuda(0).unwrap_or(Device::Cpu);

    // Load tokenizer
    let tokenizer = Tokenizer::from_file(&tokenizer_path)?;

    // Load model (GGUF format)
    let mut file = std::fs::File::open(&model_path)?;
    let content = gguf_file::Content::read(&mut file)?;
    let model = model::ModelWeights::from_gguf(content, &mut file, &device)?;

    // Initialize sampler
    let logits_processor = LogitsProcessor::new(1337, Some(0.7), Some(0.95));

    Ok(Self { model, tokenizer, device, logits_processor })
}
```

## Method: `run()` (Lines 65-96)

```

pub fn run(&mut self, prompt: &str) -> Result<String> {
    // Format prompt for grammar correction
    let formatted_prompt = format!("<bos>text\n{}\nincorrected\nn", prompt.trim());

    // Tokenize
    let tokens = self.tokenizer.encode(formatted_prompt,
true)? .get_ids().to_vec();
    let input = Tensor::new(tokens.as_slice(), &self.device)?;

    // Run model
    let logits = self.model.forward(&input, 0)?;

    // Sample next token
    let last_logits = logits.get(logits.dims2()?.0 - 1)?;
    let next_token = self.logits_processor.sample(&last_logits)?;

    // Decode
    let decoded = self.tokenizer.decode(&[next_token], true)?;

    Ok(decoded)
}

```

## ◇ `src-tauri/src/commands/` Directory

### `mod.rs` (14 lines)

**Purpose:** Re-exports all command modules.

```

mod llm;
mod misc;
mod models;
mod recording;
mod settings;
mod transcription;

pub use llm::*;
pub use misc::*;
pub use models::*;
pub use recording::*;
pub use settings::*;
pub use transcription::*;

```

### `recording.rs` (345 lines)

**Purpose:** Start and stop recording commands.

**Command: `start_recording()` (Lines 10-272)**

**Purpose:** Initialize microphone, create threads, start recording.

**Flow:**

1. Get default microphone device
2. Create WAV file path
3. Clear AI context (fresh start)
4. Create communication channels (crossbeam)
5. Spawn file writer thread
6. Spawn AI transcription thread
7. Start microphone stream
8. Store recording handle

**Command: `stop_recording()` (Lines 274-344)**

**Purpose:** Stop recording, run final transcription, cleanup.

**Flow:**

1. Take recording handle (stops mic)
  2. Wait for threads to finish
  3. For Parakeet: Return accumulated transcript
  4. For Whisper: Run VAD → High-quality final pass
  5. Clean transcript and return
- 

**`transcription.rs` (224 lines)**

**Purpose:** Sample file listing and benchmarking.

**Command: `list_sample_files()` (Lines 6-67)**

Returns available WAV files for testing.

**Command: `benchmark_test()` (Lines 69-223)**

Compares Whisper vs Parakeet performance on a sample file.

---

**`models.rs` (61 lines)**

**Purpose:** Model listing and switching.

**Commands:**

- `list_models()` - List Whisper models
- `get_current_model()` - Get active Whisper model
- `switch_model()` - Switch Whisper model
- `list_parakeet_models()` - List Parakeet models

- `init_parakeet()` - Load Parakeet model
  - `get_parakeet_status()` - Get Parakeet status
- 

## settings.rs (56 lines)

**Purpose:** Backend info and engine selection.

### Commands:

- `get_backend_info()` - Returns "CUDA", "Vulkan", or "CPU"
  - `set_active_engine()` - Switch between Whisper/Parakeet
  - `get_active_engine()` - Get currently active engine
  - `set_tray_state()` - Update tray icon color
- 

## llm.rs (98 lines)

**Purpose:** LLM initialization and inference.

### Commands:

- `init_llm()` - Load Gemma model (async, blocking task)
  - `run_llm_inference()` - Run raw inference
  - `check_llm_status()` - Check if LLM is loaded
  - `correct_text()` - Grammar correction (main function)
- 

## misc.rs (6 lines)

**Purpose:** Test command.

```
#[tauri::command]
pub fn greet(name: &str) -> String {
    format!("Hello, {}! You've been greeted from Rust!", name)
}
```

## ◇ src-tauri/src/hotkeys/ Directory

### mod.rs (3 lines)

```
mod listener;
pub use listener::start_hotkey_listener;
```

## listener.rs (75 lines)

**Purpose:** Global hotkey listener for Ctrl+Win.

```
pub fn start_hotkey_listener(app_handle: AppHandle) {
    // Atomic flags for key states
    let ctrl_held = Arc::new(AtomicBool::new(false));
    let meta_held = Arc::new(AtomicBool::new(false));
    let recording_active = Arc::new(AtomicBool::new(false));

    let callback = move |event: Event| {
        match event.event_type {
            EventType::KeyPress(key) => {
                match key {
                    Key::ControlLeft | Key::ControlRight => {
                        ctrl_held.store(true, Ordering::SeqCst);
                    }
                    Key::MetaLeft | Key::MetaRight => {
                        meta_held.store(true, Ordering::SeqCst);
                    }
                    _ => {}
                }

                // Both pressed? Start recording!
                if ctrl_held.load(...) && meta_held.load(...) &&
!recording_active.load(...) {
                    recording_active.store(true, ...);
                    app_handle.emit("hotkey-start-recording", ());
                }
            }
            EventType::KeyRelease(key) => {
                // Update flags and stop recording if released
                // ...
            }
            _ => {}
        }
    };

    // Blocks forever, listening to all keyboard events
    listen(callback).ok();
}
```

---

## src-tauri/src/tray/ Directory

mod.rs (3 lines)

```
mod icons;
pub use icons::{setup_tray, update_tray_icon};
```

icons.rs (93 lines)

**Purpose:** System tray icon management.

### Macros (Lines 7-21)

```
macro_rules! tray_icon_green {
    () => { tauri::include_image!("icons/emoji-green_circle.ico") };
}
macro_rules! tray_icon_red {
    () => { tauri::include_image!("icons/emoji-red_circle.ico") };
}
macro_rules! tray_icon_yellow {
    () => { tauri::include_image!("icons/emoji-yellow_circle.ico") };
}
```

**Function:** `update_tray_icon()` (Lines 23-50)

```
pub fn update_tray_icon(app: &AppHandle, state: AppState) -> Result<(), String> {
    let icon = match state {
        AppState::Ready => tray_icon_green!(),
        AppState::Recording => tray_icon_red!(),
        AppState::Processing => tray_icon_yellow!(),
    };

    let tooltip = match state {
        AppState::Ready => "Taurscribe - Ready",
        AppState::Recording => "Taurscribe - Recording...",
        AppState::Processing => "Taurscribe - Processing...",
    };

    if let Some(tray) = app.tray_by_id("main-tray") {
        tray.set_icon(Some(icon))?;
        tray.set_tooltip(Some(tooltip))?;
    }

    Ok(())
}
```

**Function:** `setup_tray()` (Lines 53-92)

Creates the system tray with menu items (Show, Exit).

## src-tauri/build.rs (74 lines)

**Purpose:** Build-time configuration script.

What It Does:

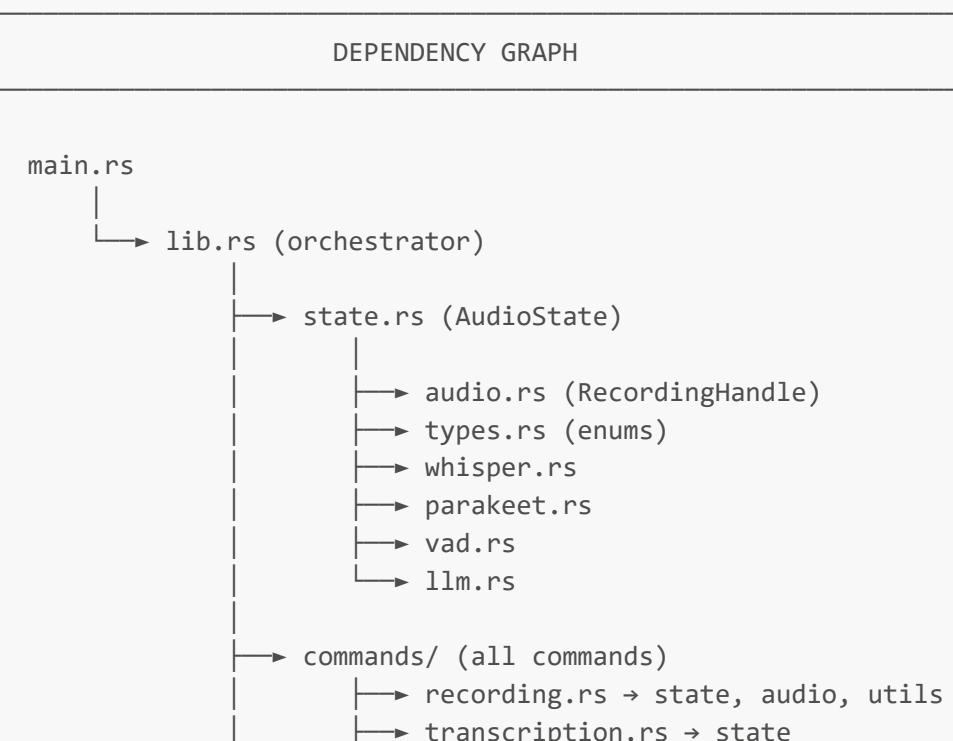
1. **Standard Tauri build:** `tauri_build::build()`
2. **macOS deployment target:** Sets `MACOSX_DEPLOYMENT_TARGET=13.4` for ONNX Runtime compatibility
3. **Windows ARM64 Clang:** Forces Clang compiler for `whisper.cpp`
4. **CUDA library paths:** Adds CUDA library search paths for Windows linker

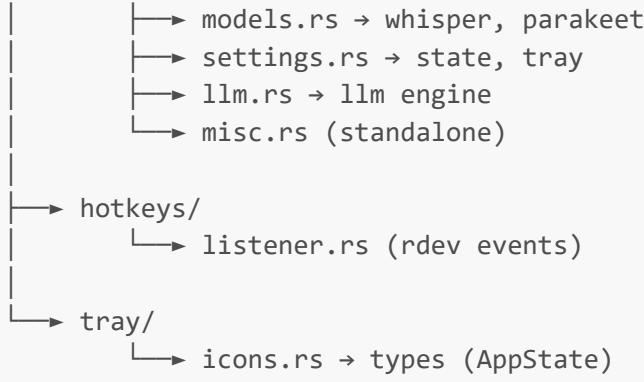
```
fn main() {
    tauri_build::build();

    #[cfg(target_os = "macos")]
    {
        println!("cargo:rustc-env=MACOSX_DEPLOYMENT_TARGET=13.4");
    }

    #[cfg(windows)]
    {
        // Find CUDA libraries for linking
        if let Ok(cuda_path) = std::env::var("CUDA_PATH") {
            let lib_path = PathBuf::from(cuda_path).join("lib").join("x64");
            if lib_path.exists() {
                println!("cargo:rustc-link-search=native={}{}", lib_path.display());
            }
        }
    }
}
```

## Complete Module Dependency Graph





## 🔗 Complete Function Reference (Every Function Explained)

This section documents **every single function** in every `.rs` file with line-by-line explanations.

### 📄 `whisper.rs` - All 15 Functions

#### Function 1: `GpuBackend::fmt()` (Lines 23-30)

**Purpose:** Implement Display trait for pretty-printing GPU backend names.

```

impl std::fmt::Display for GpuBackend {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            GpuBackend::Cuda => write!(f, "CUDA"),           // "Cuda" → "CUDA"
            GpuBackend::Vulkan => write!(f, "Vulkan"),       // "Vulkan" → "Vulkan"
            GpuBackend::Cpu => write!(f, "CPU"),             // "Cpu" → "CPU"
        }
    }
}
  
```

Parameter	Type	Description
<code>&amp;self</code>	<code>&amp;GpuBackend</code>	The enum variant to format
<code>f</code>	<code>&amp;mut Formatter</code>	The output buffer to write to
<b>Returns</b>	<code>std::fmt::Result</code>	<code>Ok(())</code> on success

#### Function 2: `null_log_callback()` (Lines 55-63)

**Purpose:** Suppress noisy C++ logs from whisper.cpp library.

```

#[cfg(target_os = "macos")]
unsafe extern "C" fn null_log_callback(
  
```

```

    _level: u32,           // Log level (ignored)
    _text: *const c_char, // Log message (ignored)
    _user_data: *mut c_void // User context (ignored)
) {
    // Do nothing - suppress all logs
}

```

Attribute	Meaning
#[cfg(target_os = "macos")]	Only compile for macOS
unsafe	Manual memory management
extern "C"	C calling convention

### Function 3: `WhisperManager::new()` (Lines 67-75)

**Purpose:** Constructor - creates a new uninitialized WhisperManager.

```

pub fn new() -> Self {
    Self {
        context: None,           // No model loaded yet
        last_transcript: String::new(), // Empty context buffer
        backend: GpuBackend::Cpu, // Assume CPU (updated on init)
        current_model: None, // No model name
        resampler: None, // Created lazily
    }
}

```

Field	Initial Value	Later Value
context	None	Some(WhisperContext) after <code>initialize()</code>
last_transcript	""	Accumulated text during session
backend	Cpu	Detected backend after <code>initialize()</code>
current_model	None	Model ID after <code>initialize()</code>
resampler	None	Created in <code>transcribe_chunk()</code>

### Function 4: `WhisperManager::get_models_dir()` (Lines 78-108)

**Purpose:** Find the directory containing Whisper model files.

```

fn get_models_dir() -> Result<std::path::PathBuf, String> {
    let possible_paths = [
        "taurscribe-runtime/models",
        "../taurscribe-runtime/models",
    ]
}

```

```

        ".../taurscribe-runtime/models",
    ];

    for path in possible_paths {
        if let Ok(canonical) = std::fs::canonicalize(path) {
            if canonical.is_dir() {
                if let Ok(entries) = std::fs::read_dir(&canonical) {
                    for entry in entries.flatten() {
                        if let Some(name) = entry.file_name().to_str() {
                            if name.starts_with("ggml-") && name.ends_with(".bin")
{
                                return Ok(canonical);
                            }
                        }
                    }
                }
            }
        }
    }

    Err("Could not find models directory containing ggml models".to_string())
}

```

### Step-by-step flow:

1. Define 3 possible relative paths to check
2. For each path:
  - o Try to canonicalize (convert to absolute path)
  - o Check if it's a directory
  - o Look for files matching `ggml-* .bin`
  - o Return immediately if found
3. Return error if all paths failed

### Function 5: `WhisperManager::list_available_models()` (Lines 110-161)

**Purpose:** Scan the models directory and return metadata for all Whisper models.

```

pub fn list_available_models() -> Result<Vec<ModelInfo>, String> {
    let models_dir = Self::get_models_dir()?;
    let mut models = Vec::new();

    let entries = std::fs::read_dir(&models_dir)
        .map_err(|e| format!("Failed to read models directory: {}", e))?;

    for entry in entries {
        let entry = entry.map_err(|e| format!("Failed to read entry: {}", e))?;
        let path = entry.path();

        if path.is_file() {
            if let Some(file_name) = path.file_name().and_then(|n| n.to_str()) {

```

```

        // Filter: ggml-*.bin, NOT silero (VAD model)
        if file_name.starts_with("ggml-")
            && file_name.ends_with(".bin")
            && !file_name.contains("silero")
        {
            let size_bytes = path.metadata().map(|m|
m.len()).unwrap_or(0);
            let size_mb = size_bytes as f32 / (1024.0 * 1024.0);

            let id = file_name
                .trim_start_matches("ggml-")
                .trim_end_matches(".bin")
                .to_string();

            let display_name = Self::format_model_name(&id);

            models.push(ModelInfo { id, display_name, file_name:
file_name.to_string(), size_mb });
        }
    }
}

models.sort_by(|a, b| a.size_mb.partial_cmp(&b.size_mb).unwrap());
Ok(models)
}

```

**Returns:** `Vec<ModelInfo>` sorted by file size (smallest first)

**Example output:**

```

[
    ModelInfo { id: "tiny.en-q5_1", display_name: "Tiny English (Q5_1)", size_mb:
42.3 },
    ModelInfo { id: "base.en-q5_1", display_name: "Base English (Q5_1)", size_mb:
78.5 },
    ModelInfo { id: "small.en-q5_1", display_name: "Small English (Q5_1)", size_mb: 245.0 },
]

```

#### Function 6: `WhisperManager::format_model_name()` (Lines 163-206)

**Purpose:** Convert cryptic model IDs to human-readable names.

```

fn format_model_name(id: &str) -> String {
    let mut name = String::new();

    // Part 1: Size

```

```

if id.contains("tiny") { name.push_str("Tiny"); }
else if id.contains("base") { name.push_str("Base"); }
else if id.contains("small") { name.push_str("Small"); }
else if id.contains("medium") { name.push_str("Medium"); }
else if id.contains("large-v3-turbo") { name.push_str("Large V3 Turbo"); }
else if id.contains("large-v3") { name.push_str("Large V3"); }
else if id.contains("large") { name.push_str("Large"); }

// Part 2: Language
if id.contains(".en") { name.push_str(" English"); }
else { name.push_str(" Multilingual"); }

// Part 3: Quantization
if id.contains("q5_0") { name.push_str(" (Q5_0)"); }
else if id.contains("q5_1") { name.push_str(" (Q5_1)"); }
else if id.contains("q8_0") { name.push_str(" (Q8_0)"); }

if name.is_empty() { return id.to_string(); }
name
}

```

**Examples:**

Input ID	Output Name
"tiny.en-q5_1"	"Tiny English (Q5_1)"
"large-v3-turbo-q8_0"	"Large V3 Turbo (Q8_0)"
"medium"	"Medium Multilingual"

**Function 7: WhisperManager::get_current_model() (Lines 208-211)**

**Purpose:** Return the currently loaded model ID.

```

pub fn get_current_model(&self) -> Option<&String> {
    self.current_model.as_ref() // Convert Option<String> to Option<&String>
}

```

**Function 8: WhisperManager::get_backend() (Lines 213-216)**

**Purpose:** Return the current GPU backend.

```

pub fn get_backend(&self) -> &GpuBackend {
    &self.backend // Return reference to avoid copying
}

```

**Function 9: `WhisperManager::clear_context()` (Lines 218-223)**

**Purpose:** Reset the accumulated transcript (used when starting new recording).

```
pub fn clear_context(&mut self) {
    self.last_transcript.clear(); // Empty the string
    println!("[INFO] Context cleared - starting fresh");
}
```

**Function 10: `WhisperManager::initialize()` (Lines 225-277)**

**Purpose:** Load a Whisper model from disk into memory and GPU.

```
pub fn initialize(&mut self, model_id: Option<&str>) -> Result<String, String> {
    // Step 1: Disable C++ logs
    unsafe { set_log_callback(None, std::ptr::null_mut()); }

    // Step 2: Find models directory
    let models_dir = Self::get_models_dir()?;

    // Step 3: Determine target model
    let target_model = model_id.unwrap_or("tiny.en-q5_1"); // Default
    let file_name = format!("ggml-{}.bin", target_model);
    let absolute_path = models_dir.join(&file_name);

    // Step 4: Verify file exists
    if !absolute_path.exists() {
        return Err(format!("Model file not found: {}", absolute_path.display()));
    }

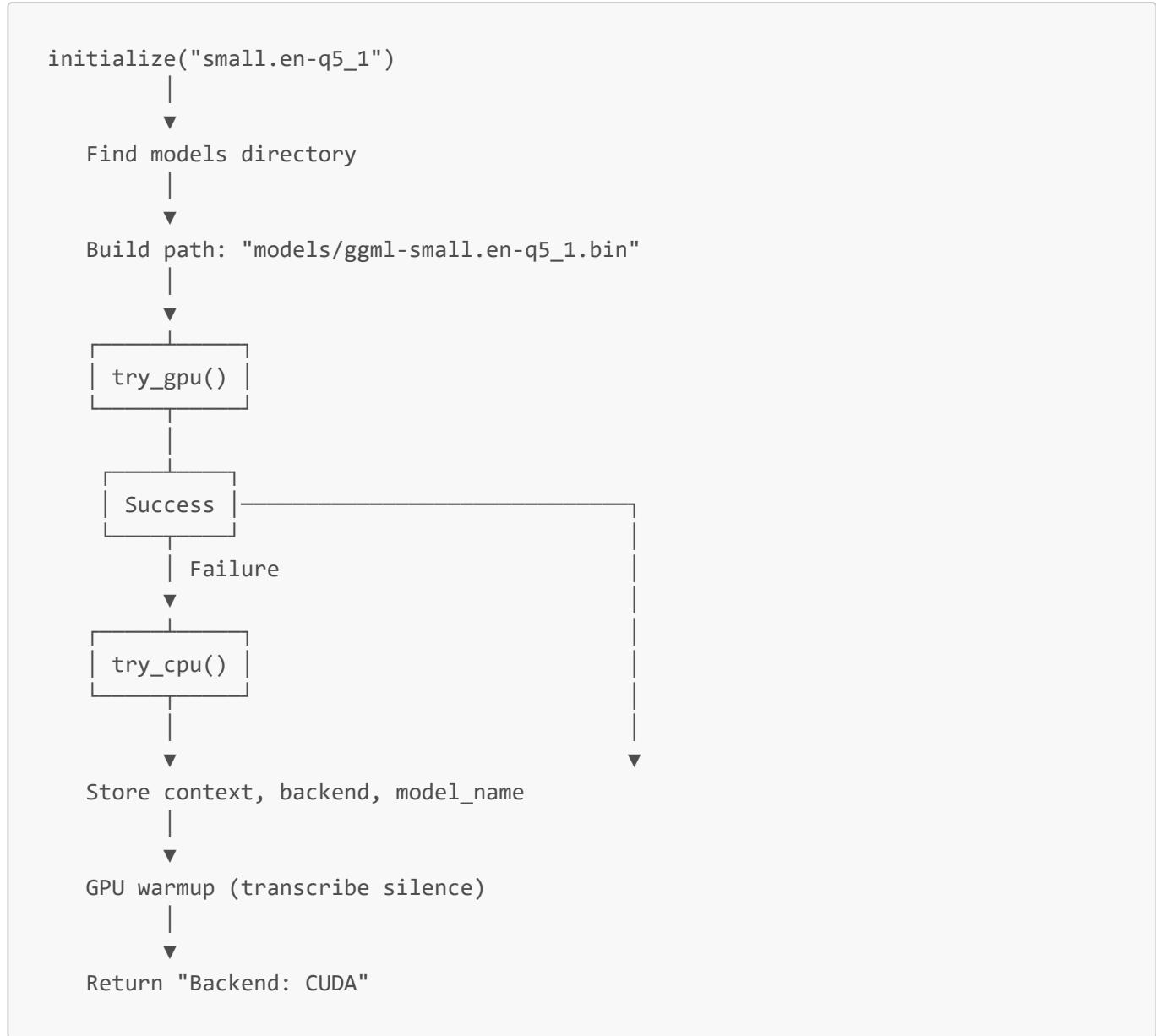
    // Step 5: Try GPU, fallback to CPU
    let (ctx, backend) = self
        .try_gpu(&absolute_path)
        .or_else(|_| self.try_cpu(&absolute_path))?;

    // Step 6: Store loaded state
    self.context = Some(ctx);
    self.backend = backend.clone();
    self.current_model = Some(target_model.to_string());

    // Step 7: GPU warmup (first inference is always slow)
    let warmup_audio = vec![0.0_f32; 16000]; // 1 second of silence
    self.transcribe_chunk(&warmup_audio, 16000).ok(); // Ignore errors

    Ok(format!("Backend: {}", backend))
}
```

**Flow diagram:**



### Function 11: `WhisperManager::try_gpu()` (Lines 279-303)

**Purpose:** Attempt to load model with GPU acceleration.

```

fn try_gpu(&self, model_path: &std::path::Path) -> Result<(WhisperContext,
GpuBackend), String> {
    println!("[GPU] Attempting GPU acceleration...");

    let mut params = WhisperContextParameters::default();
    params.use_gpu(true); // Enable GPU

    match WhisperContext::new_with_params(model_path.to_str().unwrap(), params) {
        Ok(ctx) => {
            let backend = self.detect_gpu_backend(); // CUDA or Vulkan?
            println!("[SUCCESS] ✓ GPU acceleration enabled ({})", backend);
            Ok((ctx, backend))
        }
        Err(e) => {
  
```

```
    println!("[GPU] X GPU failed: {:?}", e);
    Err(format!("GPU failed: {:?}", e))
}
}
```

**Function 12:** `WhisperManager::detect_gpu_backend()` (Lines 305-314)

**Purpose:** Determine if CUDA or Vulkan is being used.

```
fn detect_gpu_backend(&self) -> GpuBackend {
    if self.is_cuda_available() {
        return GpuBackend::Cuda; // NVIDIA detected
    }
    GpuBackend::Vulkan // Fallback assumption
}
```

**Function 13:** `WhisperManager::is_cuda_available()` (Lines 316-322)

**Purpose:** Check if NVIDIA CUDA drivers are installed.

```
fn is_cuda_available(&self) -> bool {
    std::process::Command::new("nvidia-smi") // Run nvidia-smi command
        .output()
        .map(|output| output.status.success()) // Check if it succeeded
        .unwrap_or(false) // False if command not found
}
```

#### **Function 14: WhisperManager::try_cpu() (Lines 324-341)**

**Purpose:** Fallback to CPU-only inference.

```
fn try_cpu(&self, model_path: &std::path::Path) -> Result<(WhisperContext, GpuBackend), String> {
    println!("[GPU] Falling back to CPU...");

    let params = WhisperContextParameters::default(); // No GPU flag

    match WhisperContext::new_with_params(model_path.to_str().unwrap(), params) {
        Ok(ctx) => {
            println!("[SUCCESS] ✓ CPU backend loaded");
            Ok((ctx, GpuBackend::Cpu))
        }
        Err(e) => Err(format!("Failed to load model: {:?}", e)),
    }
}
```

```

    }
}
```

### Function 15: `WhisperManager::transcribe_chunk()` (Lines 343-458)

**Purpose:** Real-time transcription of audio chunks (~6 seconds).

```

pub fn transcribe_chunk(
    &mut self,
    samples: &[f32],           // Raw audio samples
    input_sample_rate: u32,   // e.g., 48000
) -> Result<String, String> {
    // STEP 1: Get model context
    let ctx = self.context.as_mut()
        .ok_or("Whisper context not initialized")?;

    // STEP 2: Resample to 16kHz if needed
    let audio_data = if input_sample_rate != 16000 {
        // Check if resampler needs recreation
        let needs_new = match &self.resampler {
            Some((rate, size, _)) => *rate != input_sample_rate || *size != samples.len(),
            None => true,
        };

        if needs_new {
            // Create new resampler with sinc interpolation
            let params = SincInterpolationParameters {
                sinc_len: 256,
                f_cutoff: 0.95,
                interpolation: SincInterpolationType::Linear,
                window: WindowFunction::BlackmanHarris2,
                oversampling_factor: 128,
            };
            let resampler = SincFixedIn::<f32>::new(
                16000_f64 / input_sample_rate as f64, // Ratio
                2.0,                                     // Max ratio
                params,
                samples.len(),                          // Input size
                1,                                       // Channels
            ).map_err(|e| format!("Failed to create resampler: {:?}", e))?;
            self.resampler = Some((input_sample_rate, samples.len(),
Box::new(resampler)));
        }
    }

    // Run resampling
    let (_, _, resampler) = self.resampler.as_mut().unwrap();
    let waves_in = vec![samples.to_vec()];
    let waves_out = resampler.process(&waves_in, None)
        .map_err(|e| format!("Resampling failed: {:?}", e))?;
    waves_out[0].clone()
}
```

```
    } else {
        samples.to_vec()
    };

    // STEP 3: Create transcription state
    let mut state = ctx.create_state()
        .map_err(|e| format!("Failed to create state: {:?}", e))?;

    // STEP 4: Configure parameters
    let mut params = FullParams::new(SamplingStrategy::Greedy { best_of: 1 });
    params.set_n_threads(4);           // CPU threads
    params.set_translate(false);      // Don't translate
    params.set_language(Some("en"));   // English
    params.set_print_special(false);  // No special tokens
    params.set_print_progress(false); // No progress
    params.set_print_realtime(false);
    params.set_print_timestamps(false);

    // STEP 5: Add context from previous chunks
    if !self.last_transcript.is_empty() {
        params.set_initial_prompt(&self.last_transcript);
    }

    let start = std::time::Instant::now();

    // STEP 6: Run inference
    state.full(params, &audio_data)
        .map_err(|e| format!("Transcription failed: {:?}", e))?;

    // STEP 7: Extract text segments
    let num_segments = state.full_n_segments();
    let mut transcript = String::new();
    for i in 0..num_segments {
        if let Some(segment) = state.get_segment(i) {
            transcript.push_str(&segment.to_string());
        }
    }

    let final_text = transcript.trim().to_string();

    // STEP 8: Update context for next chunk
    if !final_text.is_empty() {
        if !self.last_transcript.is_empty() {
            self.last_transcript.push(' ');
        }
        self.last_transcript.push_str(&final_text);
    }

    // STEP 9: Log performance
    let duration = start.elapsed();
    let audio_duration_sec = audio_data.len() as f32 / 16000.0;
    let speedup = audio_duration_sec / duration.as_secs_f32();
    println!("[PERF] Processed {:.2}s audio in {:.0}ms | Speed: {:.1}x",
            audio_duration_sec, duration.as_millis(), speedup);
```

```
    Ok(final_text)
}
```

## Performance example:

Input: 6 seconds of audio @ 48kHz (288,000 samples)  
Step 1: Resample 48kHz → 16kHz (96,000 samples)  
Step 2: Configure Whisper params  
Step 3: Run inference (~200ms on GPU)  
Step 4: Extract text  
Output: "Hello, this is a test of the transcription system."  
Speed: 30x realtime (6s audio in 200ms)

## Function 16: `WhisperManager::transcribe_file()` (Lines 460-643)

**Purpose:** High-quality batch transcription of entire WAV file.

### 6 Steps with timing:

1. **File I/O** (~10ms): Read WAV file into memory
2. **Stereo→Mono** (~5ms): Average left/right channels
3. **Resample** (~50ms): Convert to 16kHz
4. **State Setup** (~1ms): Create Whisper state
5. **AI Inference** (~500ms): Run Whisper
6. **Extract Text** (~1ms): Get transcript segments

```
pub fn transcribe_file(&mut self, file_path: &str) -> Result<String, String> {
    // ... (565 lines of detailed processing)
    // Returns: "Full transcript of the audio file"
}
```

## Function 17: `WhisperManager::transcribe_audio_data()` (Lines 645-703)

**Purpose:** Transcribe pre-loaded audio data (used after VAD filtering).

```
pub fn transcribe_audio_data(&mut self, audio_data: &[f32]) -> Result<String, String> {
    // Same as transcribe_file but skips file loading
    // Used when audio is already in memory (after VAD filtering)
}
```

**Function 18: `WhisperManager::load_audio()` (Lines 705-779)**

**Purpose:** Load and preprocess a WAV file for VAD/Whisper.

```
pub fn load_audio(&self, file_path: &str) -> Result<Vec<f32>, String> {
    // 1. Open WAV file
    // 2. Read samples (i16 or f32)
    // 3. Convert stereo to mono
    // 4. Resample to 16kHz
    // Returns: Vec<f32> at 16kHz mono
}
```

 [parakeet.rs](#) - All 25 Functions

**Function 1: `GpuBackend::fmt()` (Lines 15-23)**

Same pattern as Whisper - formats "Cuda" → "CUDA", "DirectML" → "DirectML".

**Function 2: `ParakeetManager::new()` (Lines 60-68)**

**Purpose:** Constructor - creates uninitialized manager.

```
pub fn new() -> Self {
    ParakeetManager {
        model: None,           // No model loaded
        model_name: None,      // No model ID
        backend: GpuBackend::Cpu, // Assume CPU
        resampler: None,       // Created lazily
    }
}
```

**Function 3: `ParakeetManager::get_models_dir()` (Lines 70-97)**

Similar to Whisper but also checks executable location as fallback.

**Function 4: `ParakeetManager::list_available_models()` (Lines 99-166)**

**Purpose:** Scan for Parakeet models by detecting ONNX files.

**Detection logic:**

Files Present	Model Type
<code>encoder.onnx + decoder_joint.onnx + tokenizer.model</code>	Nemotron

Files Present	Model Type
encoder.onnx + decoder_joint.onnx + tokenizer.json	EOU
encoder.onnx + decoder.onnx + joint.onnx	TDT
model.onnx + tokenizer.json	CTC

**Function 5: ParakeetManager::estimate_model_size() (Lines 168-181)**

**Purpose:** Calculate total size of model directory in MB.

```
fn estimate_model_size(path: &PathBuf) -> f64 {
    let mut total_size = 0u64;
    if let Ok(entries) = std::fs::read_dir(path) {
        for entry in entries.flatten() {
            if let Ok(metadata) = entry.metadata() {
                if metadata.is_file() {
                    total_size += metadata.len();
                }
            }
        }
    }
    total_size as f64 / (1024.0 * 1024.0) // Convert to MB
}
```

**Function 6: ParakeetManager::get_status() (Lines 183-198)**

**Purpose:** Return current engine status for frontend.

```
pub fn get_status(&self) -> ParakeetStatus {
    ParakeetStatus {
        loaded: self.model.is_some(),
        model_id: self.model_name.clone(),
        model_type: self.model.as_ref().map(|m| match m {
            LoadedModel::Nemotron(_) => "Nemotron",
            LoadedModel::Ctc(_) => "CTC",
            LoadedModel::Eou(_) => "EOU",
            LoadedModel::Tdt(_) => "TDT",
        }).to_string(),
        backend: self.backend.to_string(),
    }
}
```

**Function 7: ParakeetManager::initialize() (Lines 200-261)**

**Purpose:** Load a Parakeet model based on type.

```

pub fn initialize(&mut self, model_id: Option<&str>) -> Result<String, String> {
    // 1. Get available models
    let available = Self::list_available_models()?;

    // 2. Select target model
    let target_id = model_id.unwrap_or(&available[0].id);
    let info = available.iter().find(|m| m.id == target_id)?;

    // 3. Parse model path from ID (e.g., "nemotron:nemotron" → "nemotron")
    let subpath = target_id.split_once(':').map(|(_, p)| p).unwrap_or(target_id);
    let model_path = models_dir.join(subpath);

    // 4. Initialize based on type
    let (model, backend) = match info.model_type.as_str() {
        "Nemotron" => { let (m, b) = Self::init_nemotron(&model_path)?;
(LoadedModel::Nemotron(m), b) }
        "CTC" => { let (m, b) = Self::init_ctc(&model_path)?;
(LoadedModel::Ctc(m), b) }
        "EOU" => { let (m, b) = Self::init_eou(&model_path)?;
(LoadedModel::Eou(m), b) }
        "TDT" => { let (m, b) = Self::init_tdt(&model_path)?;
(LoadedModel::Tdt(m), b) }
        _ => return Err("Unknown model type"),
    };

    // 5. Store state
    self.model = Some(model);
    self.model_name = Some(target_id.to_string());
    self.backend = backend;

    Ok(format!("Loaded {} ({})", info.display_name, backend))
}

```

---

**Functions 8-11:** `init_nemotron()`, `init_ctc()`, `init_eou()`, `init_tdt()` (Lines 263-471)

**Purpose:** Initialize each model type with GPU fallback chain.

**Pattern** (same for all):

```

fn init_X(path: &PathBuf) -> Result<(ModelType, GpuBackend), String> {
    #[cfg(target_os = "macos")]
    { return Ok((Self::try_cpu_X(path)?, GpuBackend::Cpu)); }

    #[cfg(not(target_os = "macos"))]
    {
        // Try CUDA first
        if let Ok(m) = Self::try_gpu_X(path) {

```

```
        return Ok((m, GpuBackend::Cuda));
    }
    // Try DirectML (Windows only)
    if let Ok(m) = Self::try_directml_X(path) {
        return Ok((m, GpuBackend::DirectML));
    }
    // Fallback to CPU
    Ok((Self::try_cpu_X(path)?, GpuBackend::Cpu))
}
```

## Functions 12-23: GPU/CPU Loaders (Lines 317-509)

12 functions for loading each model type with each backend:

Function	Model	Backend
try_gpu_nemotron()	Nemotron	CUDA
try_directml_nemotron()	Nemotron	DirectML
try_cpu_nemotron()	Nemotron	CPU
try_gpu_ctc()	CTC	CUDA
try_directml_ctc()	CTC	DirectML
try_cpu_ctc()	CTC	CPU
try_gpu_eou()	EOU	CUDA
try_directml_eou()	EOU	DirectML
try_cpu_eou()	EOU	CPU
try_gpu_tdt()	TDT	CUDA
try_directml_tdt()	TDT	DirectML
try_cpu_tdt()	TDT	CPU

**Function 24: ParakeetManager::clear_context() (Lines 513-525)**

**Purpose:** Reset model state for new recording.

```
pub fn clear_context(&mut self) {
    if let Some(model) = &mut self.model {
        match model {
            LoadedModel::Nemotron(m) => m.reset(), // Reset streaming state
            _ => {} // Other models don't need reset
        }
    }
}
```

```

    }
}
```

**Function 25: ParakeetManager::transcribe_chunk() (Lines 527-613)**

**Purpose:** Real-time transcription with model-specific handling.

```

pub fn transcribe_chunk(&mut self, samples: &[f32], sample_rate: u32) ->
Result<String, String> {
    // 1. Resample to 16kHz if needed
    let audio = if sample_rate != 16000 { /* resample */ } else { samples.to_vec() };
}

// 2. Transcribe based on model type
match model {
    LoadedModel::Nemotron(m) => {
        // Process in 560ms chunks (streaming)
        const CHUNK_SIZE: usize = 8960; // 560ms at 16kHz
        for chunk in audio.chunks(CHUNK_SIZE) {
            transcript.push_str(&m.transcribe_chunk(&chunk)?);
        }
    }
    LoadedModel::Ctc(m) => {
        // Single batch call
        let result = m.transcribe_samples(audio, 16000, 1,
Some(TimestampMode::Words))?;
        Ok(result.text)
    }
    LoadedModel::Eou(m) => {
        // 160ms chunks
        const CHUNK_SIZE: usize = 2560;
        for chunk in audio.chunks(CHUNK_SIZE) {
            full_text.push_str(&m.transcribe(&chunk, false)?);
        }
    }
    LoadedModel::Tdt(m) => {
        // Single batch call with sentence timestamps
        let result = m.transcribe_samples(audio, 16000, 1,
Some(TimestampMode::Sentences))?;
        Ok(result.text)
    }
}
```

 [vad.rs](#) - All 4 Functions

**Function 1: VADManager::new() (Lines 12-45)**

**Purpose:** Constructor with model detection.

```
pub fn new() -> Result<Self, String> {
    // Look for Silero VAD model (for future use)
    if let Ok(models_dir) = Self::get_models_dir() {
        let vad_model_path = models_dir.join("silero_vad.onnx");
        if vad_model_path.exists() {
            println!("[VAD] Found Silero VAD model");
        } else {
            println!("[VAD] Silero model not found (using energy-based VAD)");
        }
    }

    // Currently using simple energy-based VAD
    Ok(Self { threshold: 0.005 })
}
```

### Function 2: `VADManager::get_models_dir()` (Lines 47-72)

Same pattern as other modules - check 3 relative paths.

### Function 3: `VADManager::is_speech()` (Lines 74-98)

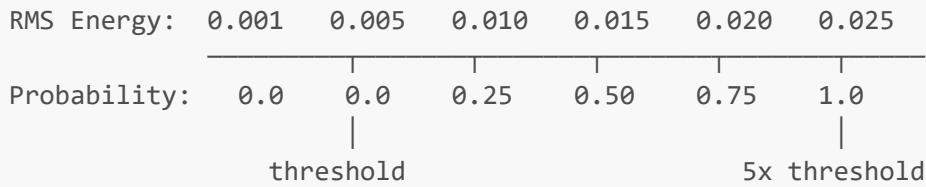
**Purpose:** Determine if audio chunk contains speech.

```
pub fn is_speech(&mut self, audio: &[f32]) -> Result<f32, String> {
    // Calculate RMS energy
    let sum_squares: f32 = audio.iter().map(|&x| x * x).sum();
    let rms = (sum_squares / audio.len() as f32).sqrt();

    // Convert to probability
    let prob = if rms < self.threshold {
        0.0 // Definitely silence
    } else if rms > self.threshold * 5.0 {
        1.0 // Definitely speech
    } else {
        // Linear interpolation between threshold and 5*threshold
        ((rms - self.threshold) / (self.threshold * 4.0)).min(1.0)
    };

    Ok(prob)
}
```

**Visualization:**



#### Function 4: `VADManager::get_speech_timestamps()` (Lines 100-213)

**Purpose:** Find speech segments in full audio file.

```
pub fn get_speech_timestamps(
    &mut self,
    audio: &[f32],
    padding_ms: usize, // Extra time around speech (500ms typical)
) -> Result<Vec<(f32, f32)>, String> {
    const FRAME_SIZE: usize = 512; // ~32ms at 16kHz
    const MIN_SPEECH_FRAMES: usize = 5; // ~150ms minimum speech

    // State machine
    let mut segments = Vec::new();
    let mut speech_start: Option<usize> = None;
    let mut consecutive_speech_frames = 0;
    let mut silence_frames = 0;

    for (i, chunk) in audio.chunks(FRAME_SIZE).enumerate() {
        let is_speech = self.is_speech(chunk)? > 0.5;

        match (is_speech, speech_start) {
            (true, None) => {
                // Speech started
                speech_start = Some(i);
                consecutive_speech_frames = 1;
            }
            (true, Some(_)) => {
                // Speech continuing
                consecutive_speech_frames += 1;
                silence_frames = 0;
            }
            (false, Some(_)) => {
                // Potential speech end
                silence_frames += 1;
                if silence_frames > padding_frames {
                    if consecutive_speech_frames >= MIN_SPEECH_FRAMES {
                        // Valid segment - save it
                        let start_time = (speech_start.unwrap() * FRAME_SIZE) as
f32 / 16000.0;
                        let end_time = (i * FRAME_SIZE) as f32 / 16000.0;
                        segments.push((start_time, end_time));
                    }
                    speech_start = None;
                }
            }
        }
    }
}
```

```

        }
    }
    (false, None) => {
        // Still silence - do nothing
    }
}

// Merge overlapping segments
let merged = merge_segments(segments);
Ok(merged)
}

```

### Example output:

```

// Input: 30 seconds of audio with speech at 2-5s and 10-15s
get_speech_timestamps(audio, 500)
// Returns: [(1.5, 5.5), (9.5, 15.5)] // With 500ms padding

```

## 📄 [llm.rs](#) - All 2 Functions

### Function 1: [LLMEngine::new\(\)](#) (Lines 19-63)

**Purpose:** Load Gemma LLM for grammar correction.

```

pub fn new() -> Result<Self> {
    // Hardcoded model path
    let base_path = PathBuf::from(
        r"c:\Users\abdul\OneDrive\Desktop\Taurscribe\taurscribe-
runtime\models\GRMR-V3-G1B-GGUF"
    );
    let model_path = base_path.join("GRMR-V3-G1B-Q4_K_M.gguf");
    let tokenizer_path = base_path.join("tokenizer.json");

    // Try CUDA, fallback to CPU
    let device = Device::new_cuda(0).unwrap_or(Device::Cpu);

    // Load tokenizer
    let tokenizer = Tokenizer::from_file(&tokenizer_path)?;

    // Load model weights (GGUF format)
    let mut file = std::fs::File::open(&model_path)?;
    let content = gguf_file::Content::read(&mut file)?;
    let model = model::ModelWeights::from_gguf(content, &mut file, &device)?;

    // Configure sampling (temperature=0.7, top_p=0.95)
    let logits_processor = LogitsProcessor::new(1337, Some(0.7), Some(0.95));
}

```

```
    Ok(Self { model, tokenizer, device, logits_processor })
}
```

**Function 2: `LLMEngine::run()` (Lines 65-96)****Purpose:** Run grammar correction inference.

```
pub fn run(&mut self, prompt: &str) -> Result<String> {
    // Format prompt for grammar model
    let formatted_prompt = format!("<bos>text\n{}\nncorrected\n", prompt.trim());

    // Tokenize
    let tokens = self.tokenizer.encode(formatted_prompt,
true)?.get_ids().to_vec();
    let input = Tensor::new(tokens.as_slice(), &self.device)?;

    // Forward pass
    let logits = self.model.forward(&input, 0)?;

    // Get last token's logits
    let (_seq_len, _vocab_size) = logits.dims2()?;
    let last_logits = logits.get(_seq_len - 1)?;

    // Sample next token
    let next_token = self.logits_processor.sample(&last_logits)?;

    // Decode
    let decoded = self.tokenizer.decode(&[next_token], true)?;

    Ok(decoded)
}
```

**`commands/recording.rs` - All 2 Functions****Function 1: `start_recording()` (Lines 12-272)****Purpose:** Initialize microphone, create threads, start recording.**Complete flow** (10 steps):

```
#[tauri::command]
pub fn start_recording(app_handle: AppHandle, state: State<AudioState>) ->
Result<String, String> {
    // STEP 1: Get microphone
    let host = cpal::default_host();
    let device = host.default_input_device().ok_or("No input device")?;
    let config: cpal::StreamConfig = device.default_input_config()?.into();
```

```
// STEP 2: Prepare output file path
let recordings_dir = get_recordings_dir()?;
let filename = format!("recording_{}.wav", chrono::Utc::now().timestamp());
let path = recordings_dir.join(&filename);

// STEP 3: Clear AI context
let active_engine = *state.active_engine.lock().unwrap();
if active_engine == ASREngine::Whisper {
    state.whisper.lock().unwrap().clear_context();
} else {
    state.parakeet.lock().unwrap().clear_context();
}

// STEP 4: Store recording path and clear transcript
*state.last_recording_path.lock().unwrap() =
Some(path.to_string_lossy().into_owned());
state.session_transcript.lock().unwrap().clear();

// STEP 5: Create WAV writer
let spec = hound::WavSpec {
    channels: config.channels,
    sample_rate: config.sample_rate.0,
    bits_per_sample: 32,
    sample_format: hound::SampleFormat::Float,
};
let writer = hound::WavWriter::create(&path, spec)?;

// STEP 6: Create communication channels
let (file_tx, file_rx) = unbounded::<Vec<f32>>();
let (whisper_tx, whisper_rx) = unbounded::<Vec<f32>>();

// STEP 7: Spawn file writer thread
std::thread::spawn(move || {
    let mut writer = writer;
    while let Ok(samples) = file_rx.recv() {
        for sample in samples {
            writer.write_sample(sample).ok();
        }
    }
    writer.finalize().ok();
});

// STEP 8: Spawn transcription thread
let whisper = state.whisper.clone();
let parakeet = state.parakeet.clone();
let vad = state.vad.clone();
let app_clone = app_handle.clone();

std::thread::spawn(move || {
    let mut buffer = Vec::new();
    let chunk_size = (sample_rate * 6) as usize; // 6 seconds

    while let Ok(samples) = whisper_rx.recv() {
```

```

        buffer.extend(samples);

        if active_engine == ASREngine::Whisper {
            // Process in 6-second chunks with VAD
            while buffer.len() >= chunk_size {
                let chunk: Vec<f32> = buffer.drain(..chunk_size).collect();
                let is_speech = vad.lock().unwrap().is_speech(&chunk)?;

                if is_speech > 0.5 {
                    let transcript = whisper.lock().unwrap()
                        .transcribe_chunk(&chunk, sample_rate)?;
                }

                if !transcript.is_empty() {
                    app_clone.emit("transcription-chunk",
TranscriptionChunk {
                    text: transcript,
                    processing_time_ms: elapsed,
                    method: "Whisper".to_string(),
                });
                }
            }
        } else {
            // Parakeet: Process in 1.12-second chunks
            let parakeet_chunk_size = (sample_rate as f32 * 1.12) as usize;
            while buffer.len() >= parakeet_chunk_size {
                let chunk: Vec<f32> =
buffer.drain(..parakeet_chunk_size).collect();
                let transcript = parakeet.lock().unwrap()
                    .transcribe_chunk(&chunk, sample_rate)?;

                if !transcript.is_empty() {
                    app_clone.emit("transcription-chunk", TranscriptionChunk {
... });
                }
            }
        });
    }

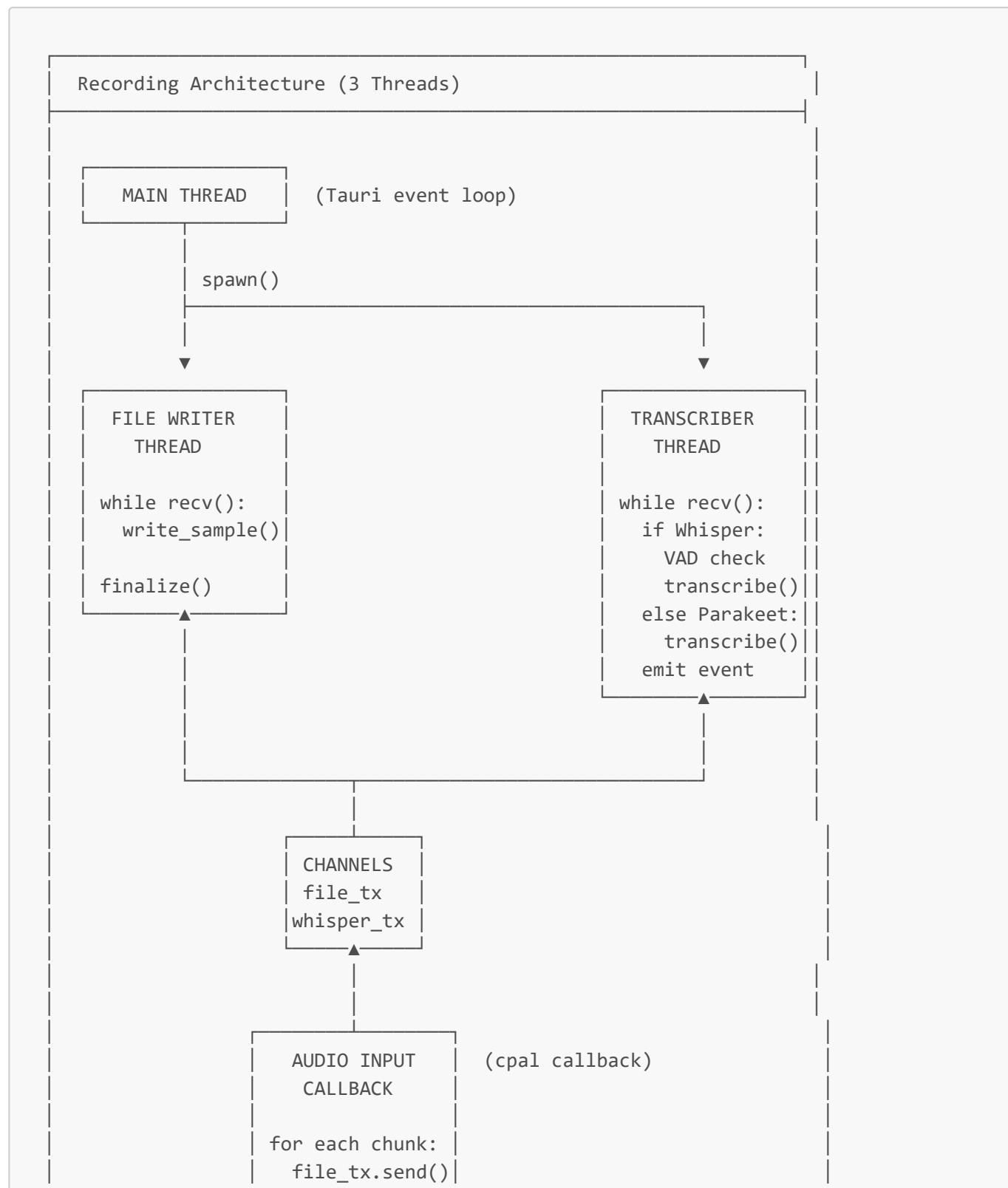
    // STEP 9: Build and start audio stream
    let stream = device.build_input_stream(
        &config,
        move |data: &[f32], _| {
            file_tx.send(data.to_vec()).ok();           // Send to file writer
            let mono = convert_to_mono(data);          // Convert stereo to mono
            whisper_tx.send(mono).ok();                // Send to transcriber
        },
        move |err| eprintln!("Audio input error: {}", err),
        None,
    )?;
    stream.play()?;
}

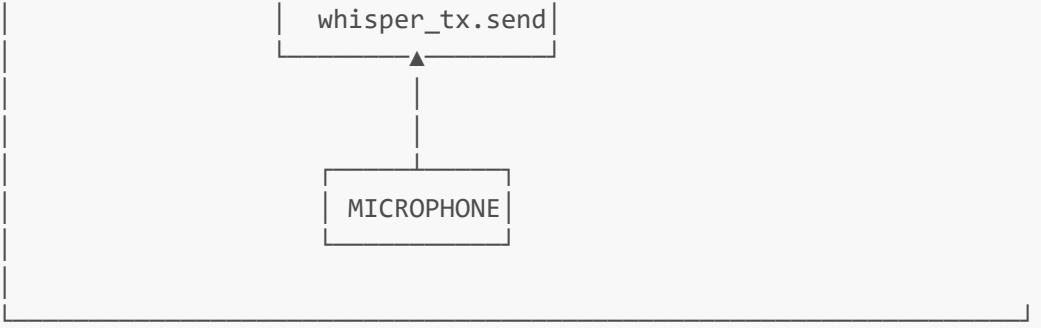
// STEP 10: Store recording handle

```

```
*state.recording_handle.lock().unwrap() = Some(RecordingHandle {  
    stream: SendStream(stream),  
    file_tx,  
    whisper_tx,  
});  
  
Ok(format!("Recording started: {}", path.display()))  
}
```

## Thread architecture:





## Function 2: `stop_recording()` (Lines 274-344)

**Purpose:** Stop recording, run final transcription, cleanup.

```

#[tauri::command]
pub fn stop_recording(state: State<AudioState>) -> Result<String, String> {
    // STEP 1: Take recording handle (stops mic)
    let mut handle = state.recording_handle.lock().unwrap();
    if let Some(recording) = handle.take() {
        // STEP 2: Drop stream and channels (signals threads to stop)
        drop(recording.stream);
        drop(recording.file_tx);
        drop(recording.whisper_tx);

        // STEP 3: Wait for threads to finish
        std::thread::sleep(std::time::Duration::from_millis(500));

        let active_engine = *state.active_engine.lock().unwrap();

        // STEP 4A: For Parakeet - use accumulated transcript
        if active_engine == ASREngine::Parakeet {
            let transcript = state.session_transcript.lock().unwrap().clone();
            let final_text = clean_transcript(&transcript);
            return Ok(final_text);
        }

        // STEP 4B: For Whisper - run high-quality final pass
        let path = state.last_recording_path.lock().unwrap().clone();
        if let Some(path) = path {
            // Load audio file
            let mut whisper = state.whisper.lock().unwrap();
            let audio_data = whisper.load_audio(&path)?;

            // Apply VAD filtering
            let mut vad = state.vad.lock().unwrap();
            let timestamps = vad.get_speech_timestamps(&audio_data, 500)?;

            if timestamps.is_empty() {
                return Ok("[silence]".to_string());
            }
        }
    }
}

```

```

// Extract speech segments only
let mut clean = Vec::new();
for (start, end) in timestamps {
    let s = (start * 16000.0) as usize;
    let e = (end * 16000.0) as usize;
    clean.extend_from_slice(&audio_data[s..e]);
}

// Final transcription
let result = whisper.transcribe_audio_data(&clean)?;
Ok(result)
} else {
    Ok("Recording saved.".to_string())
}
} else {
    Err("Not recording".to_string())
}
}

```

## 📄 commands/transcription.rs - All 2 Functions

### Function 1: `list_sample_files()` (Lines 6-67)

**Purpose:** List WAV files available for benchmarking.

```

#[tauri::command]
pub fn list_sample_files() -> Result<Vec<SampleFile>, String> {
    let possible_paths = [
        "taurscribe-runtime/samples",
        "../taurscribe-runtime/samples",
        "../../taurscribe-runtime/samples",
    ];
    // Find directory with .wav files
    let mut target_dir = PathBuf::new();
    for path in possible_paths {
        if let Ok(p) = std::fs::canonicalize(path) {
            if p.is_dir() && has_wav_files(&p) {
                target_dir = p;
                break;
            }
        }
    }
    // List all .wav files
    let mut files = Vec::new();
    for entry in std::fs::read_dir(target_dir)? {
        if let Ok(entry) = entry {
            let path = entry.path();
            if path.extension().map(|e| e == "wav").unwrap_or(false) {
                files.push(SampleFile {

```

```

        name: path.file_name().unwrap().to_string_lossy().to_string(),
        path: path.to_string_lossy().to_string(),
    });
}
}

files.sort_by(|a, b| a.name.cmp(&b.name));
Ok(files)
}

```

**Function 2: benchmark_test() (Lines 69-223)**

**Purpose:** Compare Whisper vs Parakeet performance.

```

#[tauri::command]
pub fn benchmark_test(state: State<AudioState>, file_path: String) ->
Result<String, String> {
    // 1. Load audio file
    let mut reader = hound::WavReader::open(&absolute_path)?;
    let audio_duration_secs = sample_count / sample_rate / channels;

    // 2. Convert to mono samples
    let mono_samples = convert_to_mono(samples);

    // 3. Test Whisper (without VAD)
    state.whisper.lock().unwrap().clear_context();
    let start_whisper_naive = Instant::now();
    for chunk in mono_samples.chunks(chunk_size) {
        state.whisper.lock().unwrap().transcribe_chunk(chunk, sample_rate).ok()?;
    }
    let time_whisper_naive = start_whisper_naive.elapsed();

    // 4. Test Whisper (with VAD)
    state.whisper.lock().unwrap().clear_context();
    let start_whisper_vad = Instant::now();
    let mut chunks_skipped = 0;
    for chunk in mono_samples.chunks(chunk_size) {
        let is_speech = state.vad.lock().unwrap().is_speech(chunk)?;
        if is_speech > 0.5 {
            state.whisper.lock().unwrap().transcribe_chunk(chunk,
sample_rate).ok()?;
        } else {
            chunks_skipped += 1;
        }
    }
    let time_whisper_vad = start_whisper_vad.elapsed();

    // 5. Test Parakeet
    let start_parakeet = Instant::now();
    for chunk in mono_samples.chunks(parakeet_chunk_size) {

```

```

        state.parakeet.lock().unwrap().transcribe_chunk(chunk, sample_rate).ok();
    }
    let time_parakeet = start_parakeet.elapsed();

    // 6. Calculate speed factors
    let factor_whisper = audio_duration_secs / time_whisper_vad.as_secs_f32();
    let factor_parakeet = audio_duration_secs / time_parakeet.as_secs_f32();

    // 7. Determine winner
    let winner = if time_whisper_vad < time_parakeet { "Whisper AI" } else {
        "NVIDIA Parakeet" };

    // 8. Format results
    Ok(format!(
        "📊 BENCHMARK RESULTS\n\
        🎙 WHISPER AI:\n\
        - Baseline: {:.2}s\n\
        - Optimized: {:.2}s\n\
        - Speed: {:.1}x\n\n\
        🐦 PARAKEET:\n\
        - Streaming: {:.2}s\n\
        - Speed: {:.1}x\n\n\
        🏆 WINNER: {}",
        time_whisper_naive.as_secs_f32(),
        time_whisper_vad.as_secs_f32(),
        factor_whisper,
        time_parakeet.as_secs_f32(),
        factor_parakeet,
        winner
    ))
}
}

```

## 📄 commands/models.rs - All 6 Functions

### Function 1: `list_models()` (Lines 7-11)

```

#[tauri::command]
pub fn list_models() -> Result<Vec<whisper::ModelInfo>, String> {
    whisper::WhisperManager::list_available_models()
}

```

### Function 2: `get_current_model()` (Lines 13-18)

```

#[tauri::command]
pub fn get_current_model(state: State<AudioState>) -> Result<Option<String>, String> {
    let whisper = state.whisper.lock().unwrap();

```

```
    Ok(whisper.get_current_model().cloned())
}
```

### Function 3: `switch_model()` (Lines 20-35)

```
#[tauri::command]
pub fn switch_model(state: State<AudioState>, model_id: String) -> Result<String, String> {
    // Safety check
    let handle = state.recording_handle.lock().unwrap();
    if handle.is_some() {
        return Err("Cannot switch models while recording".to_string());
    }
    drop(handle);

    let mut whisper = state.whisper.lock().unwrap();
    whisper.initialize(Some(&model_id))
}
```

### Function 4: `list_parakeet_models()` (Lines 37-41)

```
#[tauri::command]
pub fn list_parakeet_models() -> Result<Vec<parakeet::ParakeetModelInfo>, String> {
    parakeet::ParakeetManager::list_available_models()
}
```

### Function 5: `init_parakeet()` (Lines 43-53)

```
#[tauri::command]
pub fn init_parakeet(state: State<AudioState>, model_id: Option<String>) -> Result<String, String> {
    let mut parakeet = state.parakeet.lock().unwrap();
    let result = parakeet.initialize(model_id.as_deref())?;

    // Auto-switch to Parakeet
    *state.active_engine.lock().unwrap() = ASREngine::Parakeet;

    Ok(result)
}
```

### Function 6: `get_parakeet_status()` (Lines 55-60)

```
#[tauri::command]
pub fn get_parakeet_status(state: State<AudioState>) ->
Result<parakeet::ParakeetStatus, String> {
    let parakeet = state.parakeet.lock().unwrap();
    Ok(parakeet.get_status())
}
```

## commands/settings.rs - All 4 Functions

### Function 1: get_backend_info() (Lines 7-11)

```
#[tauri::command]
pub fn get_backend_info(state: State<AudioState>) -> Result<String, String> {
    let whisper = state.whisper.lock().unwrap();
    Ok(format!("{}", whisper.get_backend())) // "CUDA", "Vulkan", or "CPU"
}
```

### Function 2: set_active_engine() (Lines 13-25)

```
#[tauri::command]
pub fn set_active_engine(state: State<AudioState>, engine: String) ->
Result<String, String> {
    let new_engine = match engine.to_lowercase().as_str() {
        "whisper" => ASREngine::Whisper,
        "parakeet" => ASREngine::Parakeet,
        _ => return Err(format!("Unknown engine: {}", engine)),
    };

    *state.active_engine.lock().unwrap() = new_engine;
    Ok(format!("Engine switched to {:?}", new_engine))
}
```

### Function 3: get_active_engine() (Lines 27-31)

```
#[tauri::command]
pub fn get_active_engine(state: State<AudioState>) -> Result<ASREngine, String> {
    Ok(*state.active_engine.lock().unwrap())
}
```

### Function 4: set_tray_state() (Lines 33-55)

```

#[tauri::command]
pub fn set_tray_state(
    app: AppHandle,
    state: State<AudioState>,
    new_state: String,
) -> Result<(), String> {
    let app_state = match new_state.as_str() {
        "ready" => AppState::Ready,
        "recording" => AppState::Recording,
        "processing" => AppState::Processing,
        _ => return Err(format!("Unknown state: {}", new_state)),
    };

    *state.current_app_state.lock().unwrap() = app_state;
    tray::update_tray_icon(&app, app_state)?;

    Ok(())
}

```

## commands/llm.rs - All 4 Functions

### Function 1: `init_llm()` (Lines 5-34)

```

#[tauri::command]
pub async fn init_llm(state: State<'_, AudioState>) -> Result<String, String> {
    // Check if already loaded
    {
        let llm_guard = state.llm.lock().unwrap();
        if llm_guard.is_some() {
            return Ok("LLM already initialized".to_string());
        }
    }

    // Load in blocking task (heavy operation)
    let result = tauri::async_runtime::spawn_blocking(move || LLMEngine::new())
        .await
        .map_err(|e| format!("JoinError: {}", e))?;

    match result {
        Ok(engine) => {
            let mut llm_guard = state.llm.lock().unwrap();
            *llm_guard = Some(engine);
            Ok("Gemma LLM initialized successfully".to_string())
        }
        Err(e) => Err(format!("Failed to load LLM: {}", e)),
    }
}

```

**Function 2: `run_llm_inference()` (Lines 36-65)**

```
#[tauri::command]
pub async fn run_llm_inference(
    state: State<'_, AudioState>,
    prompt: String,
) -> Result<String, String> {
    let llm_handle = state.llm.clone();

    let output = tauri::async_runtime::spawn_blocking(move || {
        let mut llm_guard = llm_handle.lock().unwrap();
        if let Some(engine) = llm_guard.as_mut() {
            engine.run(&prompt).map_err(|e| e.to_string())
        } else {
            Err("LLM not initialized".to_string())
        }
    })
    .await??;

    Ok(output)
}
```

**Function 3: `check_llm_status()` (Lines 67-71)**

```
#[tauri::command]
pub fn check_llm_status(state: State<'_, AudioState>) -> bool {
    state.llm.lock().unwrap().is_some()
}
```

**Function 4: `correct_text()` (Lines 73-97)**

```
#[tauri::command]
pub async fn correct_text(state: State<'_, AudioState>, text: String) ->
Result<String, String> {
    let llm_handle = state.llm.clone();

    let output = tauri::async_runtime::spawn_blocking(move || {
        let mut llm_guard = llm_handle.lock().unwrap();
        if let Some(engine) = llm_guard.as_mut() {
            engine.run(&text).map_err(|e| e.to_string())
        } else {
            Err("LLM not initialized. Please load Gemma first.".to_string())
        }
    })
    .await??;
```

```
    Ok(output)
}
```

## 📄 commands/misc.rs - 1 Function

### Function 1: greet() (Lines 2-5)

```
#[tauri::command]
pub fn greet(name: &str) -> String {
    format!("Hello, {}! You've been greeted from Rust!", name)
}
```

## 📄 hotkeys/listener.rs - 1 Function

### Function 1: start_hotkey_listener() (Lines 6-74)

**Purpose:** Listen for Ctrl+Win global hotkey to start/stop recording.

```
pub fn start_hotkey_listener(app_handle: AppHandle) {
    // Atomic flags for key states
    let ctrl_held = Arc::new(AtomicBool::new(false));
    let meta_held = Arc::new(AtomicBool::new(false));
    let recording_active = Arc::new(AtomicBool::new(false));

    let callback = move |event: Event| {
        match event.event_type {
            EventType::KeyPress(key) => {
                // Track key presses
                match key {
                    Key::ControlLeft | Key::ControlRight => {
                        ctrl_held.store(true, Ordering::SeqCst);
                    }
                    Key::MetaLeft | Key::MetaRight => {
                        meta_held.store(true, Ordering::SeqCst);
                    }
                    _ => {}
                }

                // Both pressed? Start recording!
                if ctrl_held.load(Ordering::SeqCst)
                    && meta_held.load(Ordering::SeqCst)
                    && !recording_active.load(Ordering::SeqCst)
                {
                    recording_active.store(true, Ordering::SeqCst);
                    app_handle.emit("hotkey-start-recording", ());
                }
            }
        }
    }
}
```

```

EventType::KeyRelease(key) => {
    // Track key releases
    match key {
        Key::ControlLeft | Key::ControlRight => {
            ctrl_held.store(false, Ordering::SeqCst);
        }
        Key::MetaLeft | Key::MetaRight => {
            meta_held.store(false, Ordering::SeqCst);
        }
        _ => {}
    }

    // Either released? Stop recording!
    if recording_active.load(Ordering::SeqCst)
        && (!ctrl_held.load(Ordering::SeqCst) ||
!meta_held.load(Ordering::SeqCst))
    {
        recording_active.store(false, Ordering::SeqCst);
        app_handle.emit("hotkey-stop-recording", ());
    }
    _ => {}
}
};

// This blocks forever - must be run in separate thread
listen(callback).ok();
}

```

## tray/icons.rs - 2 Functions

### Function 1: update_tray_icon() (Lines 23-50)

**Purpose:** Change system tray icon based on app state.

```

pub fn update_tray_icon(app: &AppHandle, state: AppState) -> Result<(), String> {
    // Select icon based on state
    let icon = match state {
        AppState::Ready => tray_icon_green!(), // ○
        AppState::Recording => tray_icon_red!(), // ⚡
        AppState::Processing => tray_icon_yellow!(), // ○
    };

    // Select tooltip text
    let tooltip = match state {
        AppState::Ready => "Taurscribe - Ready",
        AppState::Recording => "Taurscribe - Recording...",
        AppState::Processing => "Taurscribe - Processing...",
    };

    // Apply changes
}

```

```

if let Some(tray) = app.tray_by_id("main-tray") {
    tray.set_icon(Some(icon))?;
    tray.set_tooltip(Some(tooltip))?;
}

Ok(())
}

```

## Function 2: `setup_tray()` (Lines 53-92)

**Purpose:** Create system tray with menu.

```

pub fn setup_tray(app: &tauri::App) -> Result<(), Box<dyn std::error::Error>> {
    // Create menu items
    let show_item = MenuItem::with_id(app, "show", "Show Taurscribe", true, None:<&str>)?;
    let quit_item = MenuItem::with_id(app, "quit", "Exit", true, None:<&str>)?;
    let separator = PredefinedMenuItem::separator(app)?;
    let menu = Menu::with_items(app, [&show_item, &separator, &quit_item])?;

    // Build tray icon
    let _tray = TrayIconBuilder::with_id("main-tray")
        .icon(tray_icon_green!())
        .tooltip("Taurscribe - Ready")
        .menu(&menu)
        .show_menu_on_left_click(false)
        .on_menu_event(|app, event| {
            match event.id.as_ref() {
                "show" => {
                    if let Some(window) = app.get_webview_window("main") {
                        window.show();
                        window.set_focus();
                    }
                }
                "quit" => app.exit(0),
                _ => {}
            }
        })
        .on_tray_icon_event(|tray, event| {
            if let TrayIconEvent::Click { .. } = event {
                // Left-click shows window
                if let Some(window) = tray.app_handle().get_webview_window("main") {
                    window.show();
                    window.set_focus();
                }
            }
        })
        .build(app)?;
}

```

```
    Ok(())
}
```

## (utils.rs) - 2 Functions

### Function 1: `clean_transcript()` (Lines 1-31)

Already documented above.

### Function 2: `get_recordings_dir()` (Lines 33-46)

Already documented above.

## (state.rs) - 1 Function

### Function 1: `AudioState::new()` (Lines 40-54)

Already documented above.

## (build.rs) - 1 Function

### Function 1: `main()` (Lines 1-73)

**Purpose:** Build-time configuration for CUDA and platform-specific settings.

```
fn main() {
    // Standard Tauri build
    tauri_build::build();

    // macOS: Set deployment target for ONNX Runtime
    #[cfg(target_os = "macos")]
    {
        println!("cargo:rustc-env=MACOSX_DEPLOYMENT_TARGET=13.4");
    }

    // Windows ARM64: Force Clang compiler
    #[cfg(all(target_os = "windows", target_arch = "aarch64"))]
    {
        std::env::set_var("CC", "clang-cl");
        std::env::set_var("CXX", "clang-cl");
    }

    // Windows: Add CUDA library search paths
    #[cfg(windows)]
    {
        if let Ok(cuda_path) = std::env::var("CUDA_PATH") {
            let lib_path = PathBuf::from(cuda_path).join("lib").join("x64");
            if lib_path.exists() {

```

```
        println!("cargo:rustc-link-search=native={}", lib_path.display());
    }
}
}
```

## Function Count Summary

File	Functions	Lines
whisper.rs	18	781
parakeet.rs	25	615
vad.rs	4	215
llm.rs	2	98
lib.rs	1	119
main.rs	1	7
state.rs	1	55
types.rs	0 (types only)	31
utils.rs	2	47
audio.rs	0 (types only)	17
commands/recording.rs	2	345
commands/transcription.rs	2	224
commands/models.rs	6	61
commands/settings.rs	4	56
commands/llm.rs	4	98
commands/misc.rs	1	6
hotkeys/listener.rs	1	75
tray/icons.rs	2	93
build.rs	1	74
<b>TOTAL</b>	<b>77 functions</b>	<b>3,017 lines</b>

## Analogies for Complex Concepts

This section explains the trickiest parts of the codebase using real-world analogies.

- ## 1. Arc<Mutex<T>> - The Shared Notebook in a Library

## The Code:

```
pub struct AudioState {
    pub whisper: Arc<Mutex<WhisperManager>>,
    pub vad: Arc<Mutex<VADManager>>,
    // ...
}
```

## The Analogy: A Shared Notebook in a Library Study Room

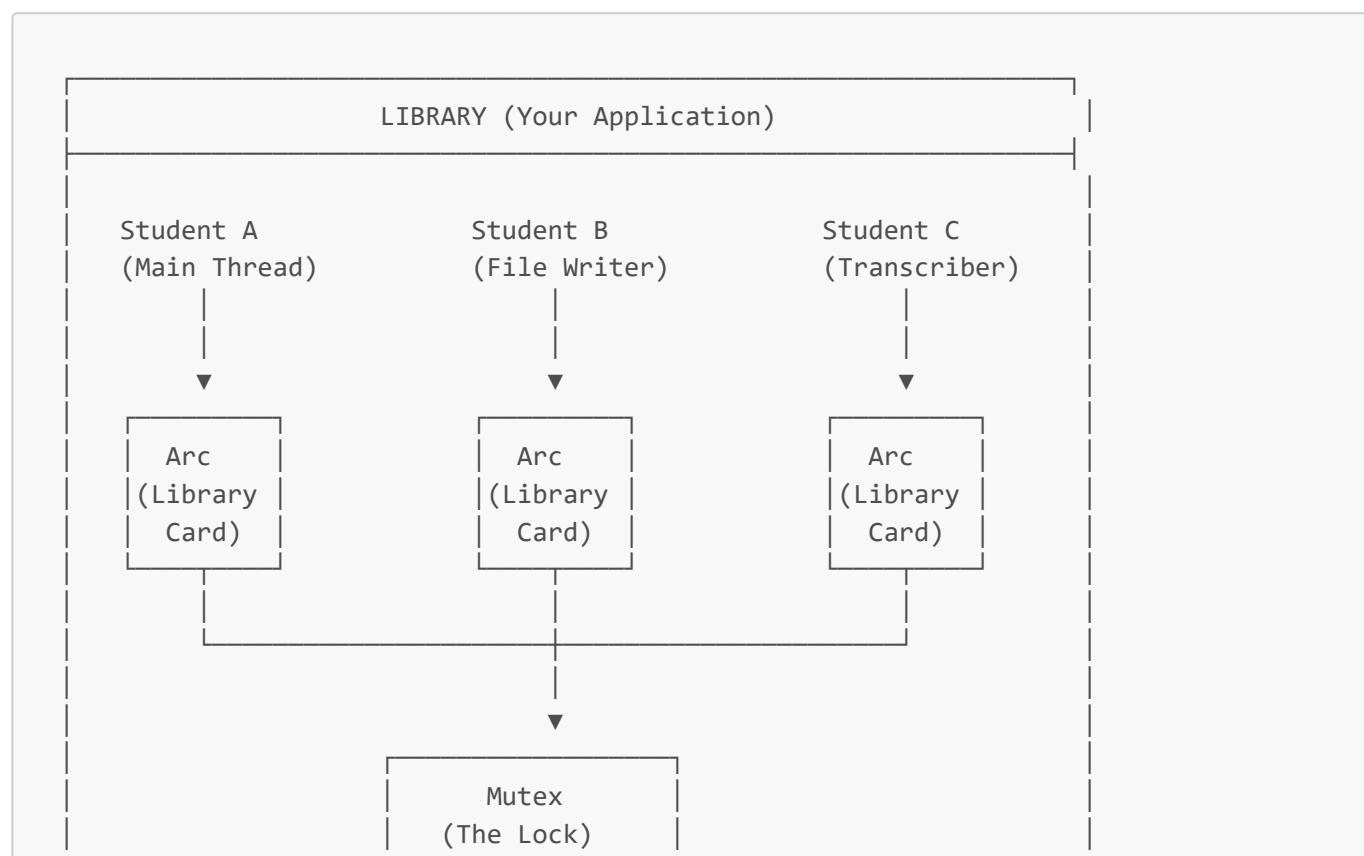
Imagine a library study room where multiple students need to use the same notebook:

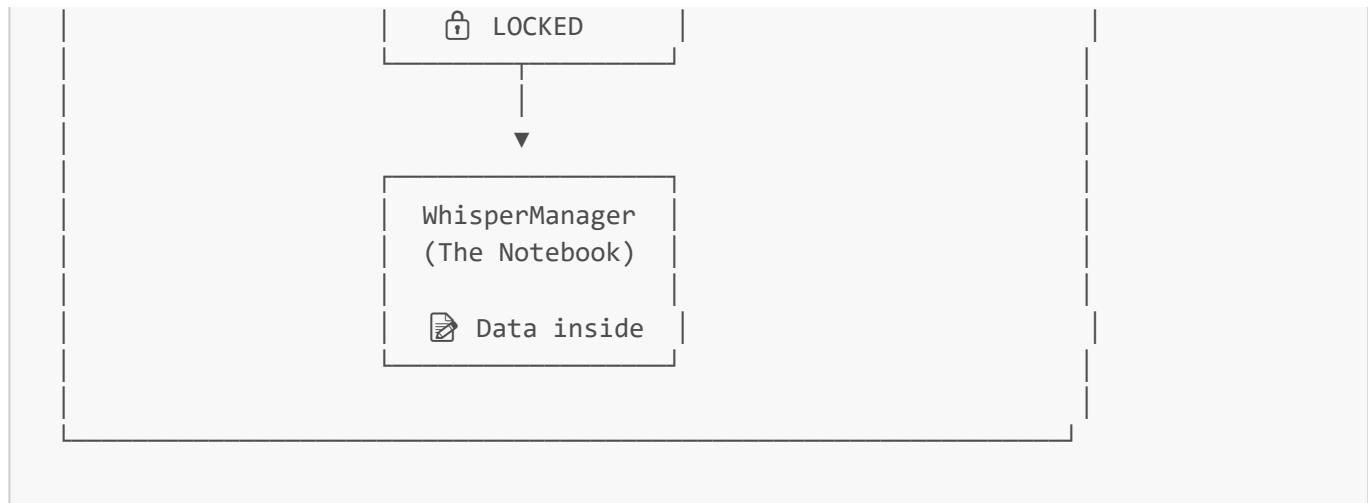
Rust Concept	Library Analogy
WhisperManager	The notebook itself (contains valuable information)
Mutex<...>	A lock on the notebook - only one person can write at a time
Arc<...>	Multiple library cards that all point to the same notebook
.lock().unwrap()	"I'd like to check out the notebook, please"
drop(guard)	Returning the notebook so others can use it

## Why we need both:

- **Without Mutex:** Two students write at the same time → scribbled mess (data corruption)
- **Without Arc:** Each student gets their own copy → changes aren't shared

## Visual:





## 2. Channels (`crossbeam_channel`) - The Sushi Conveyor Belt

### The Code:

```

let (file_tx, file_rx) = unbounded::<Vec<f32>>();
let (whisper_tx, whisper_rx) = unbounded::<Vec<f32>>();

// Sender side (in audio callback)
file_tx.send(data.to_vec()).ok();

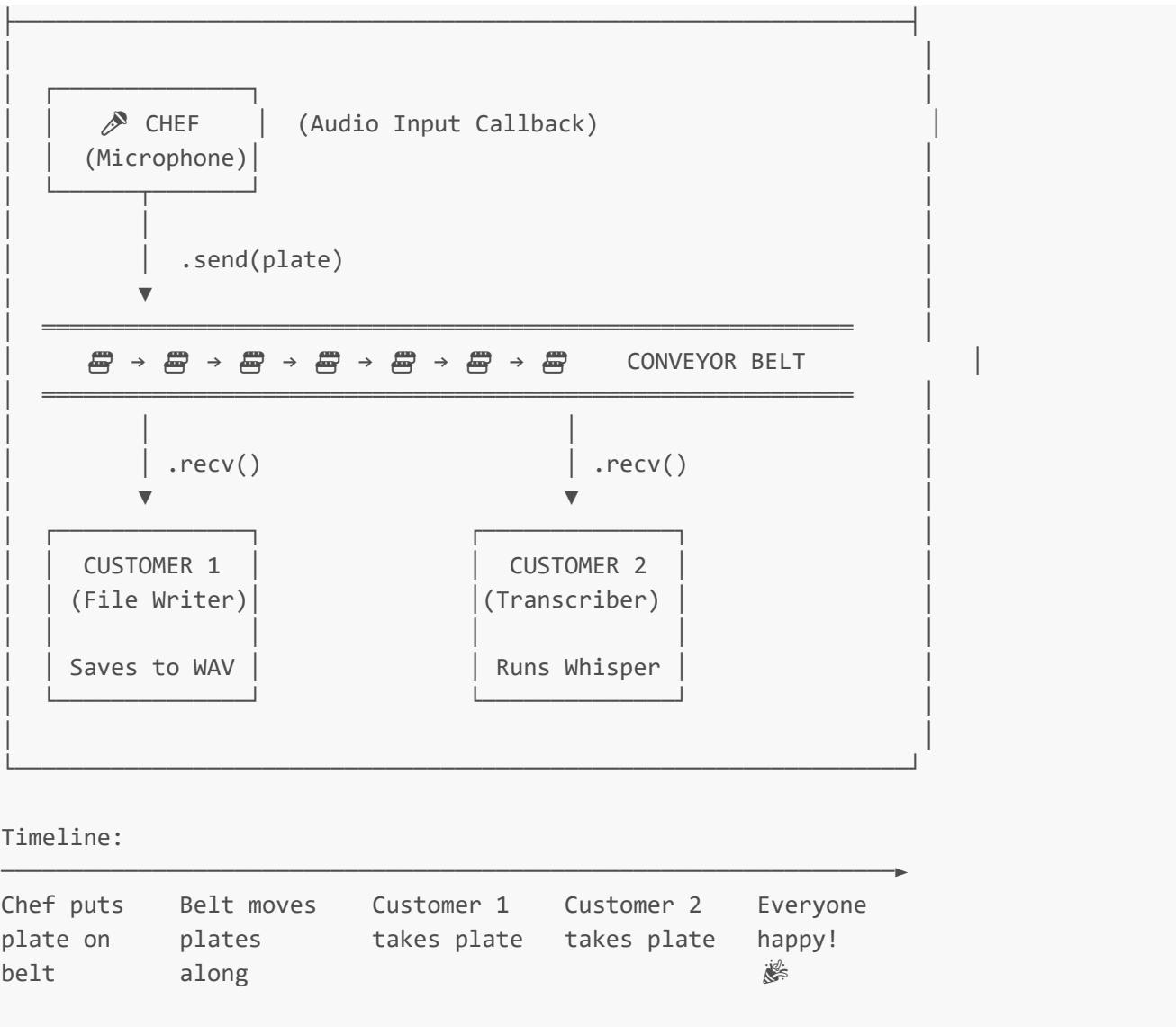
// Receiver side (in file writer thread)
while let Ok(samples) = file_rx.recv() {
    // process samples
}
  
```

### The Analogy: A Sushi Restaurant Conveyor Belt

Rust Concept	Sushi Analogy
<code>unbounded()</code>	Creating a new conveyor belt
<code>tx</code> (transmitter)	The chef putting plates on the belt
<code>rx</code> (receiver)	The customer taking plates off the belt
<code>Vec&lt;f32&gt;</code>	A plate of sushi (audio samples)
<code>.send()</code>	Chef places a plate on the belt
<code>.recv()</code>	Customer picks up a plate
<code>unbounded</code>	Belt can hold unlimited plates (no backpressure)

### Visual:





## Why this pattern?

- Chef doesn't wait for customers (non-blocking)
- Customers don't fight over plates (separate belts)
- Belt can buffer plates if customers are slow

## 3. Audio Resampling - The Currency Exchange

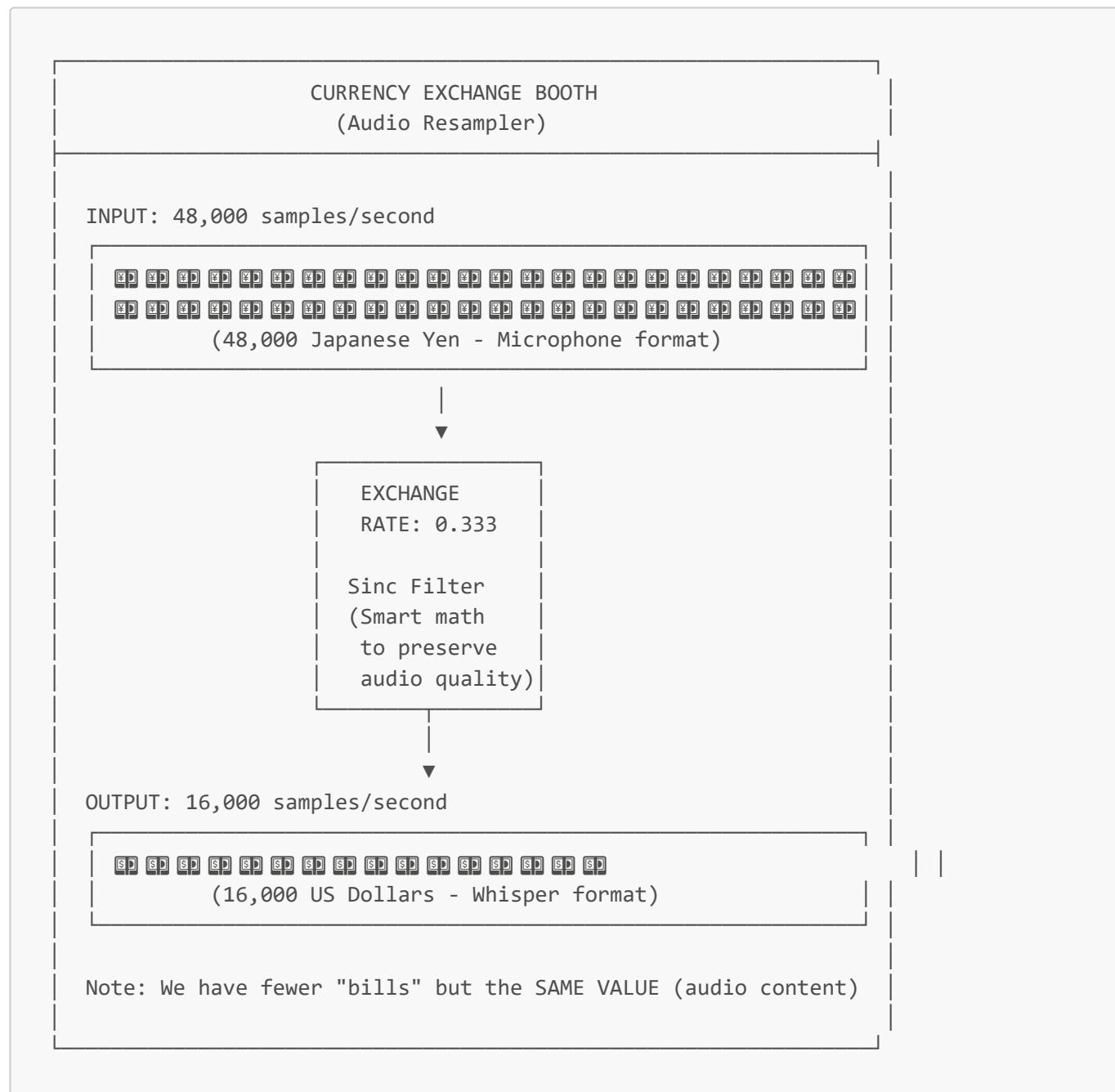
### The Code:

```

let resampler = SincFixedIn::<f32>::new(
    16000_f64 / input_sample_rate as f64, // Ratio: 16000/48000 = 0.333
    2.0,
    params,
    samples.len(),
    1,
)?;
let resampled = resampler.process(&[samples], None)?;
  
```

### The Analogy: Exchanging Currency at the Airport

Rust Concept	Currency Analogy
48000 Hz audio	48,000 Japanese Yen
16000 Hz audio	16,000 US Dollars (what Whisper accepts)
SincFixedIn	The currency exchange booth
$16000/48000 = 0.333$	Exchange rate (1 USD = 3 JPY)
sinc interpolation	Sophisticated math to avoid losing value
<code>samples.len()</code>	How many bills you're exchanging

**Visual:****Why 16kHz?**

- Whisper was trained on 16kHz audio (it only speaks "US Dollars")

- Human speech is mostly below 8kHz (Nyquist: 16kHz captures this perfectly)
- Higher sample rates waste processing power with inaudible frequencies

## 4. VAD (Voice Activity Detection) - The Security Guard

### The Code:

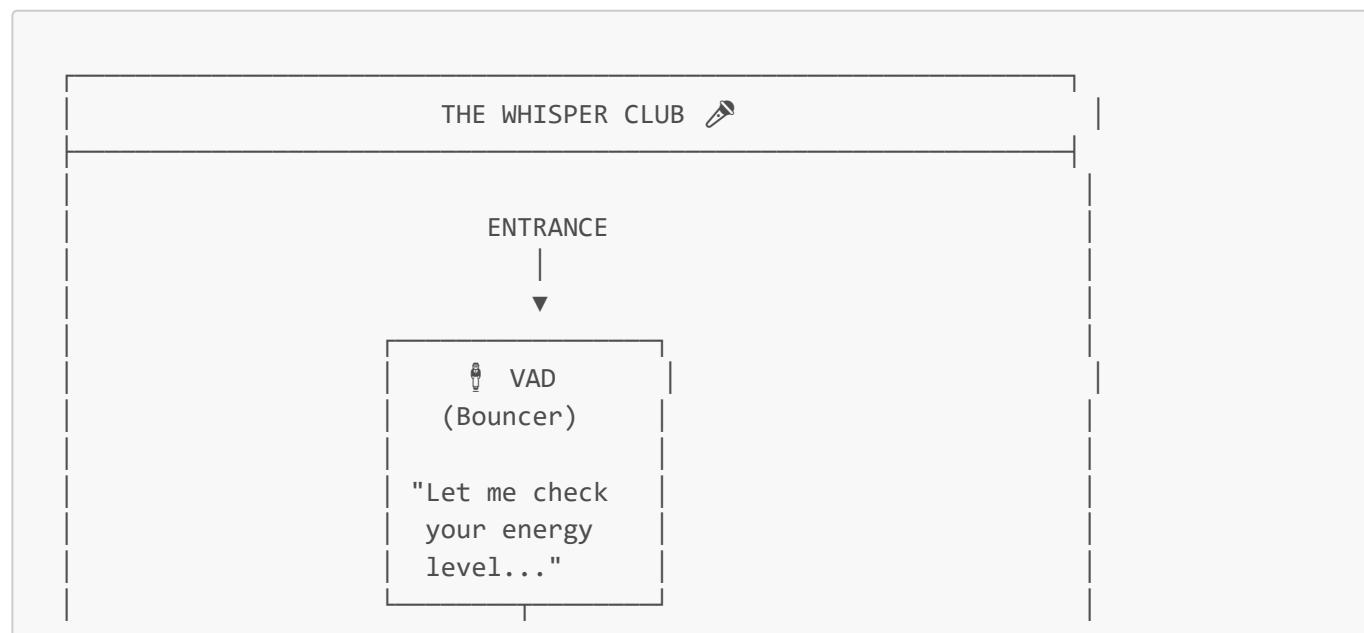
```
pub fn is_speech(&mut self, audio: &[f32]) -> Result<f32, String> {
    let sum_squares: f32 = audio.iter().map(|&x| x * x).sum();
    let rms = (sum_squares / audio.len() as f32).sqrt();

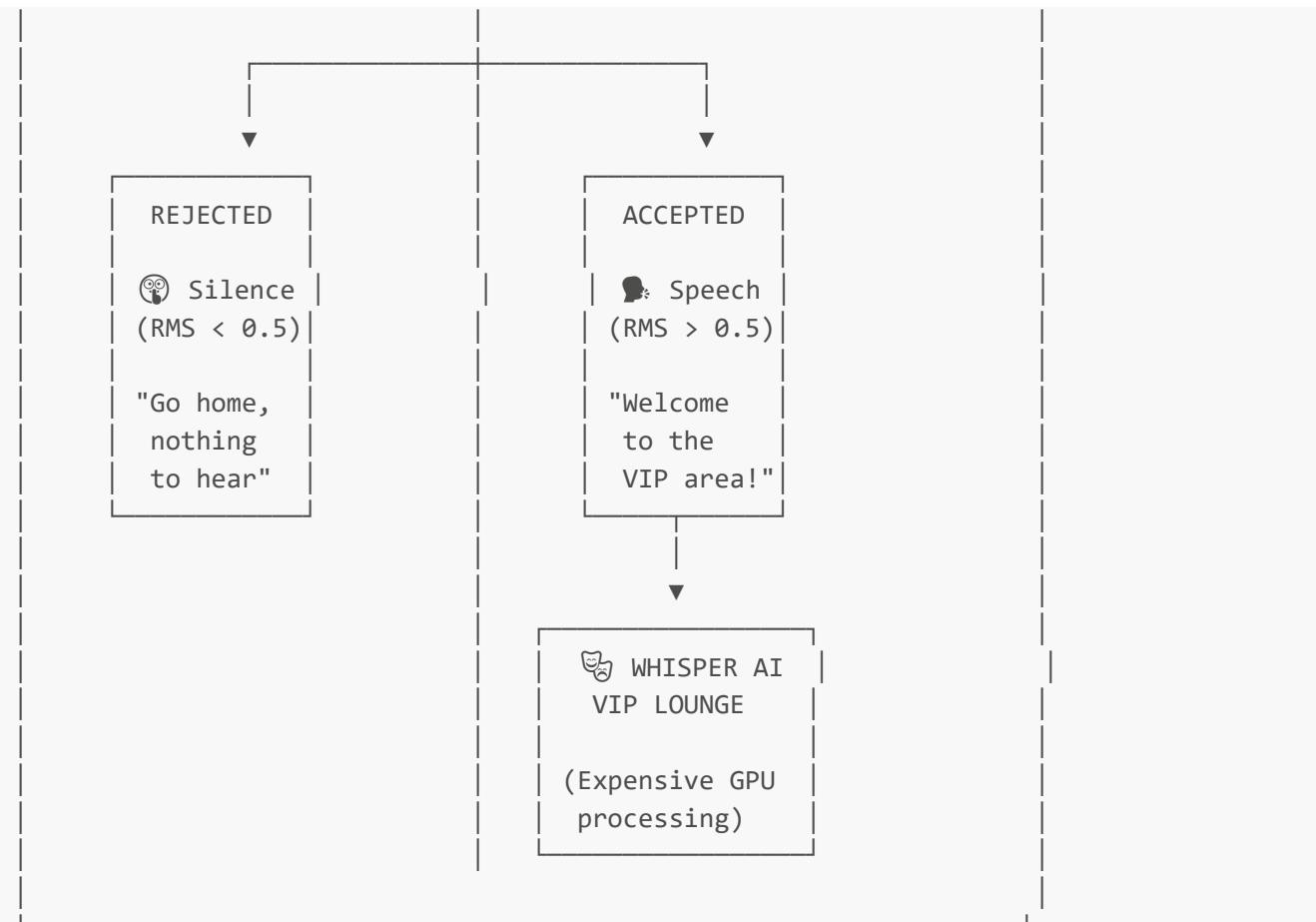
    if rms < self.threshold { 0.0 }      // Silence
    else if rms > self.threshold * 5.0 { 1.0 } // Definitely speech
    else { /* interpolate */ }
}
```

### The Analogy: A Security Guard at a VIP Club

Rust Concept	Club Analogy
VAD	The bouncer at the door
audio chunk	A person trying to enter
RMS energy	How loudly they're dressed (flashy = high energy)
threshold	Dress code minimum
> 0.5 (speech)	"You may enter the Whisper VIP lounge"
< 0.5 (silence)	"Sorry, too quiet. Wait outside."
Whisper	The expensive VIP area (costs GPU time)

### Visual:





WITHOUT VAD:

Process EVERYTHING

- Silence: GPU: 100% load
- Slow
- Wasteful

WITH VAD:

Only process SPEECH chunks

- Silence: GPU: 30% load
- Fast
- Efficient

## Why VAD matters:

- 60-70% of a typical recording is silence
- Whisper AI is expensive (GPU time = money)
- VAD = Free filter that saves 60-70% of processing

## 5. GPU Backends - The Transportation Options

### The Code:

```

pub enum GpuBackend {
    Cuda, // NVIDIA
    Vulkan, // AMD/Intel (Whisper)
    DirectML, // Windows (Parakeet)
    Cpu, // Fallback
}
  
```

```

}

// Initialization tries them in order:
let (ctx, backend) = self.try_gpu(&path)
    .or_else(|_| self.try_cpu(&path))?;

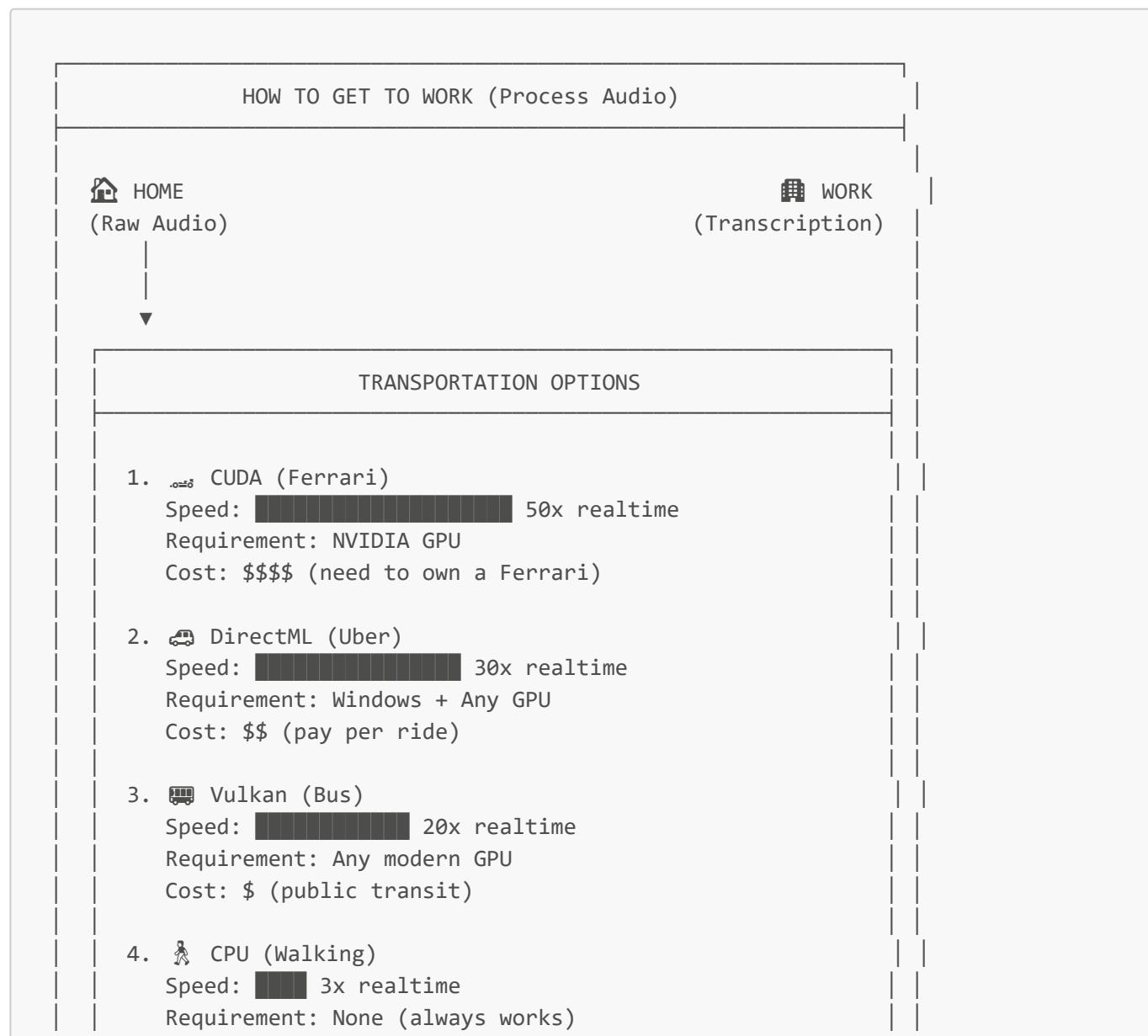
```

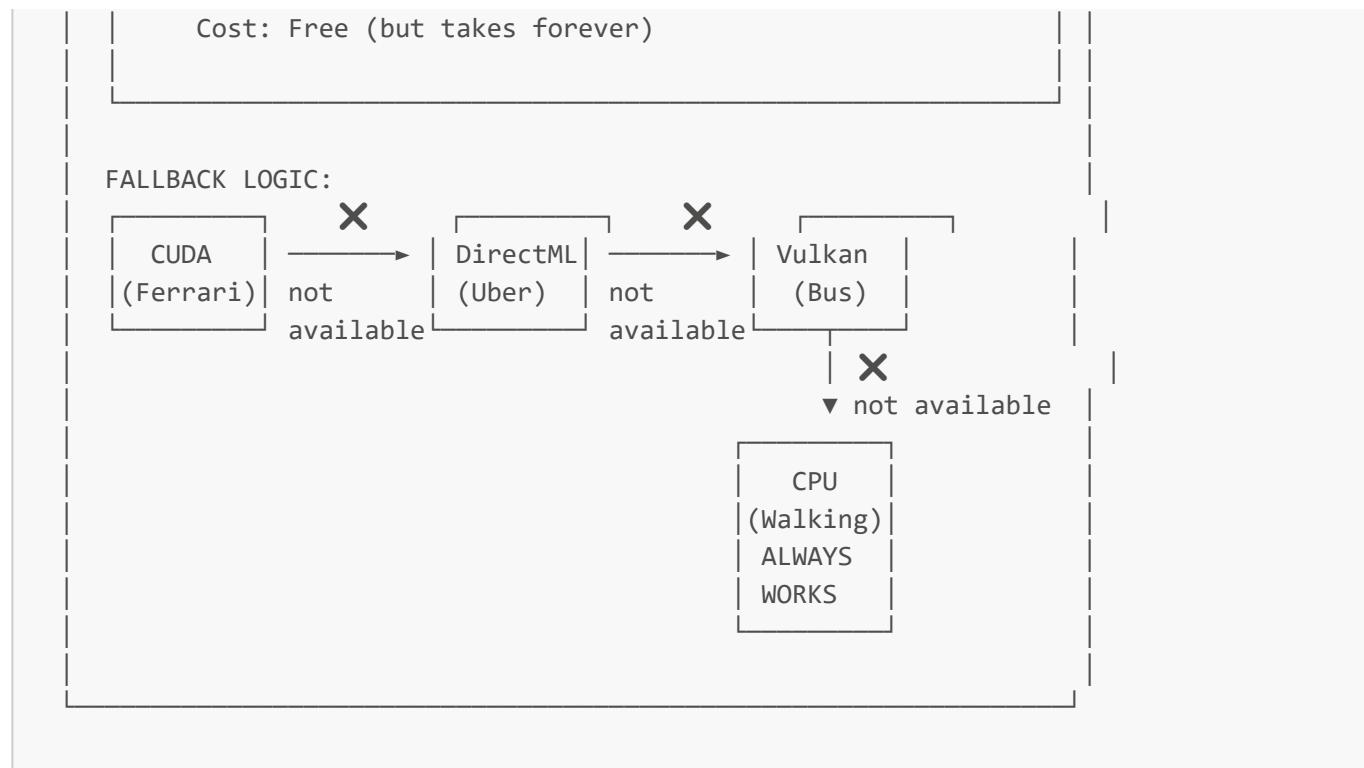
## The Analogy: Getting to Work

### Rust Concept    Transportation Analogy

Cpu	Walking (slow but always works)
Vulkan	Taking the bus (decent, widely available)
DirectML	Taking an Uber (good on Windows)
Cuda	Taking a Ferrari (fastest, but need to own one)
or_else	"If Ferrari unavailable, try Uber, then bus, then walk"

### Visual:





## 6. Whisper Context/Prompting - The Conversation Memory

### The Code:

```

// Store previous text
if !final_text.is_empty() {
    self.last_transcript.push_str(&final_text);
}

// Use it as context for next chunk
if !self.last_transcript.is_empty() {
    params.set_initial_prompt(&self.last_transcript);
}
  
```

### The Analogy: A Stenographer Taking Notes

Rust Concept	Stenographer Analogy
<code>last_transcript</code>	The stenographer's notepad
<code>set_initial_prompt()</code>	"Here's what we discussed so far..."
Whisper AI	The stenographer
Each audio chunk	A new sentence spoken
Context	Memory of the conversation

### Visual:

## COURTROOM STENOGRAPHER (Whisper AI)

## WITHOUT CONTEXT:

Speaker: "...and that's why I went to the BANK."

Stenographer: "BANK? 🤔"

- River bank? (nature)
- Money bank? (finance)
- Blood bank? (medical)

Result: "...and that's why I went to the BLANK." ✗

## WITH CONTEXT:

Notepad (last_transcript):

"I needed to deposit my paycheck. The ATM was broken, so I had to go inside..."

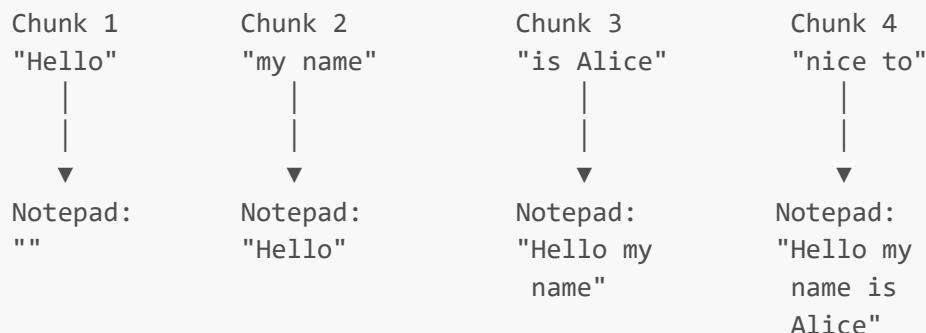
Speaker: "...and that's why I went to the BANK."

Stenographer: "BANK! 🌟"

- Context mentions paycheck, ATM, deposit
- Obviously a MONEY bank!

Result: "...and that's why I went to the bank." ✓

## TIMELINE:



## Why context matters:

- Whisper processes audio in chunks (not the whole file at once)
- Each chunk is independent by default
- Giving it previous text helps it understand ambiguous words
- Dramatically improves accuracy for names, technical terms, etc.

## 7. Option<T> and Result<T, E> - The Package Delivery

### The Code:

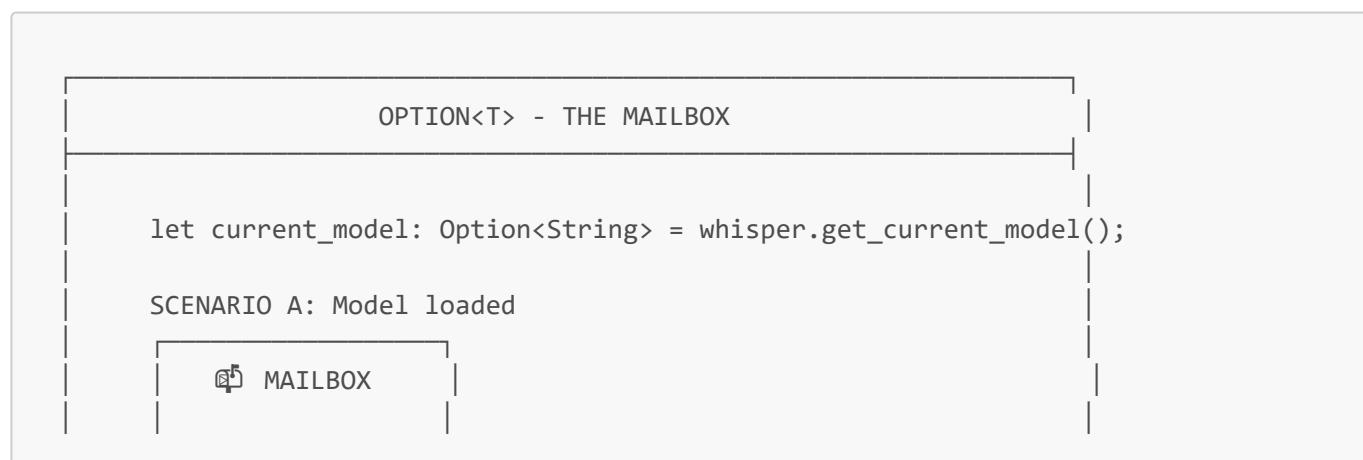
```
// Option: Maybe there's something inside
let model: Option<String> = whisper.get_current_model().cloned();

// Result: Either success or failure
let transcript: Result<String, String> = whisper.transcribe_chunk(&audio, 16000);
```

### The Analogy: Checking Your Mailbox

Rust Concept	Mailbox Analogy
Option<T>	A mailbox (might have mail, might be empty)
Some(value)	"You've got mail! 📬"
None	"Mailbox is empty 🚫"
Result<T, E>	A tracked package delivery
Ok(value)	"Package delivered successfully! 📦"
Err(error)	"Delivery failed: Address not found 🚫"
.unwrap()	"I'm 100% sure there's mail, just give it to me"
.ok_or()	"If empty, pretend this error happened"
? operator	"If error, stop and return it to caller"

### Visual:



```
Some(  
    "tiny.en"  
)
```

← There's a model name inside!

SCENARIO B: No model loaded

```
MAILBOX
```

```
None
```

← Nothing here yet

HANDLING IT:

```
match current_model {  
    Some(name) => println!("Model: {}", name), // Use it!  
    None => println!("No model loaded"), // Handle it  
}
```

RESULT<T, E> - PACKAGE DELIVERY

```
let transcript: Result<String, String> =  
    whisper.transcribe_chunk(&audio, 16000);
```

SCENARIO A: Success!

```
DELIVERED
```

```
Ok(  
    "Hello  
    world"  
)
```

← Transcription succeeded!

SCENARIO B: Failure!

```
FAILED
```

```
Err(  
    "Model  
    not  
    loaded"  
)
```

← Something went wrong

```
HANDLING IT:
match transcript {
    Ok(text) => println!("Heard: {}", text),
    Err(e) => eprintln!("Error: {}", e),
}

OR with ? operator (propagate error to caller):
let text = whisper.transcribe_chunk(&audio, 16000)?;
// If Err, function returns early with that error
// If Ok, unwraps and continues
```

## 8. **unsafe** and **Send/Sync** - The Safety Waiver

### The Code:

```
pub struct SendStream(pub cpal::Stream);

// "I promise this is safe to send between threads"
unsafe impl Send for SendStream {}
unsafe impl Sync for SendStream {}
```

### The Analogy: Signing a Liability Waiver at an Adventure Park

Rust Concept	Adventure Park Analogy
Rust's type system	The safety rules (helmets, harnesses, etc.)
<b>Send</b> trait	"Can this equipment be passed to another person?"
<b>Sync</b> trait	"Can multiple people use this at the same time?"
<b>unsafe impl</b>	Signing the waiver: "I take responsibility"
<b>cpal::Stream</b>	A specialized piece of equipment
The compiler	The safety inspector

### Visual:

ADVENTURE PARK SAFETY SYSTEM

NORMAL EQUIPMENT (Rust-approved):

String, Vec<f32>, etc.

- Send: Can pass to another person
- Sync: Multiple people can reference

Safety Inspector: "All good! ✓"

#### SPECIAL EQUIPMENT (cpal::Stream):

cpal::Stream (Audio hardware handle)

- ? Send: Compiler doesn't know if safe
- ? Sync: Compiler doesn't know if safe

Safety Inspector: "I can't verify this... 🤷"

YOU (the developer):

##### 📄 LIABILITY WAIVER

I, the developer, hereby declare that:

```
unsafe impl Send for SendStream {}
unsafe impl Sync for SendStream {}
```

I have verified that cpal::Stream:

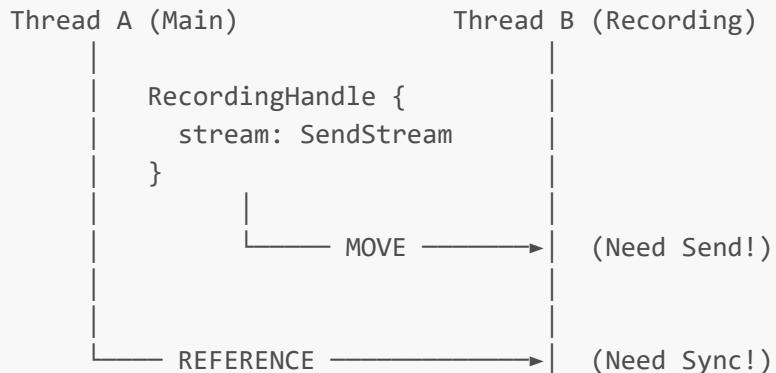
- Can be safely moved between threads
- Can be safely shared via &reference

Signature: _Developer_Name_

Safety Inspector: "Okay, you signed the waiver. ✓"

"But if something breaks, it's on you!"

#### WHY WE NEED THIS:



#### Why this is necessary:

- `cpal::Stream` is from an external C library
- Rust doesn't know if it's safe to share between threads
- We (the developers) verified it's safe for our use case
- `unsafe` = "Trust me, I know what I'm doing"

## 9. The Recording Thread Architecture - The Restaurant Kitchen

### The Code (simplified):

```
// Main thread sets up the kitchen
let (order_tx, order_rx) = unbounded(); // Order system

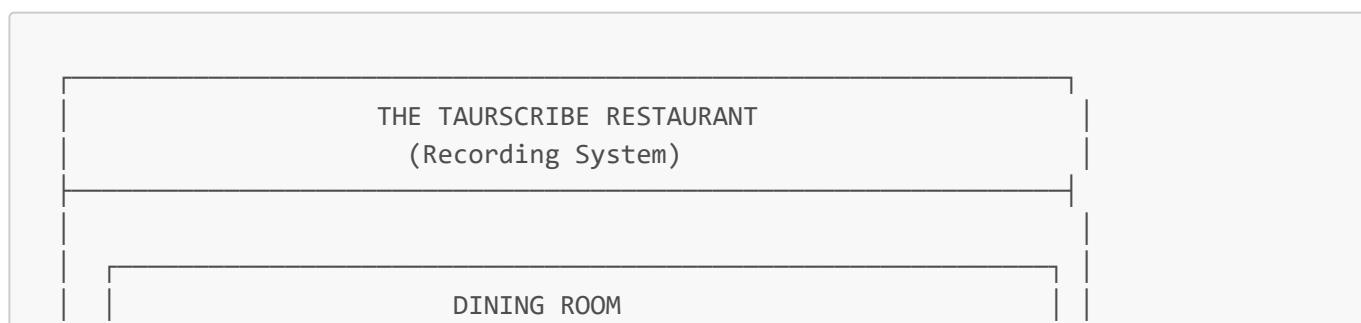
// Spawn chef thread
std::thread::spawn(move || {
    while let Ok(order) = order_rx.recv() {
        cook(order);
    }
});

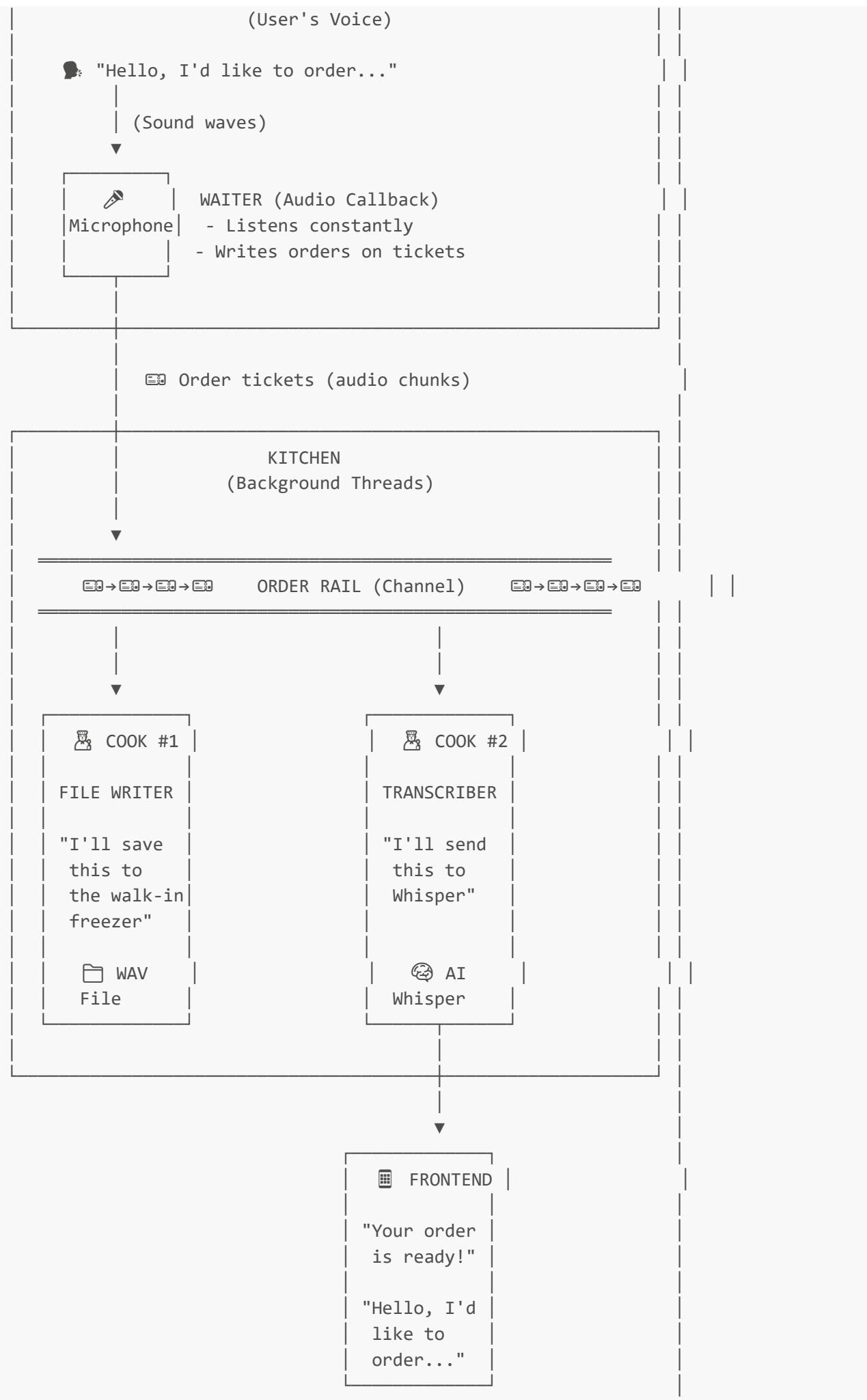
// Waiter takes orders
stream.on_data(|food| {
    order_tx.send(food).ok();
});
```

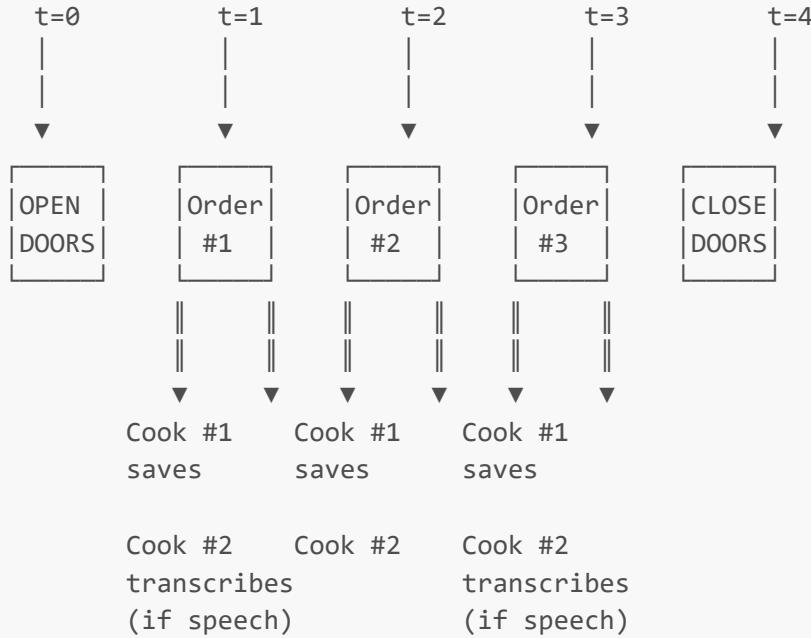
### The Analogy: A Busy Restaurant Kitchen

Rust Concept	Restaurant Analogy
Main thread	Restaurant manager
Audio callback	Waiter taking orders
File writer thread	Line cook #1 (saves to storage)
Transcriber thread	Line cook #2 (processes orders)
Channels	Order tickets on a rail
<code>stream.play()</code>	Opening for business
<code>drop(tx)</code>	Closing the kitchen

### Visual:





**TIMELINE:****Why multiple threads?**

- Waiter can't stop for chefs (audio callback must be fast)
- Chefs work independently (file writing doesn't block AI)
- Order rail prevents lost tickets (channel buffers data)
- Closing doors signals kitchen to finish up (dropping tx)

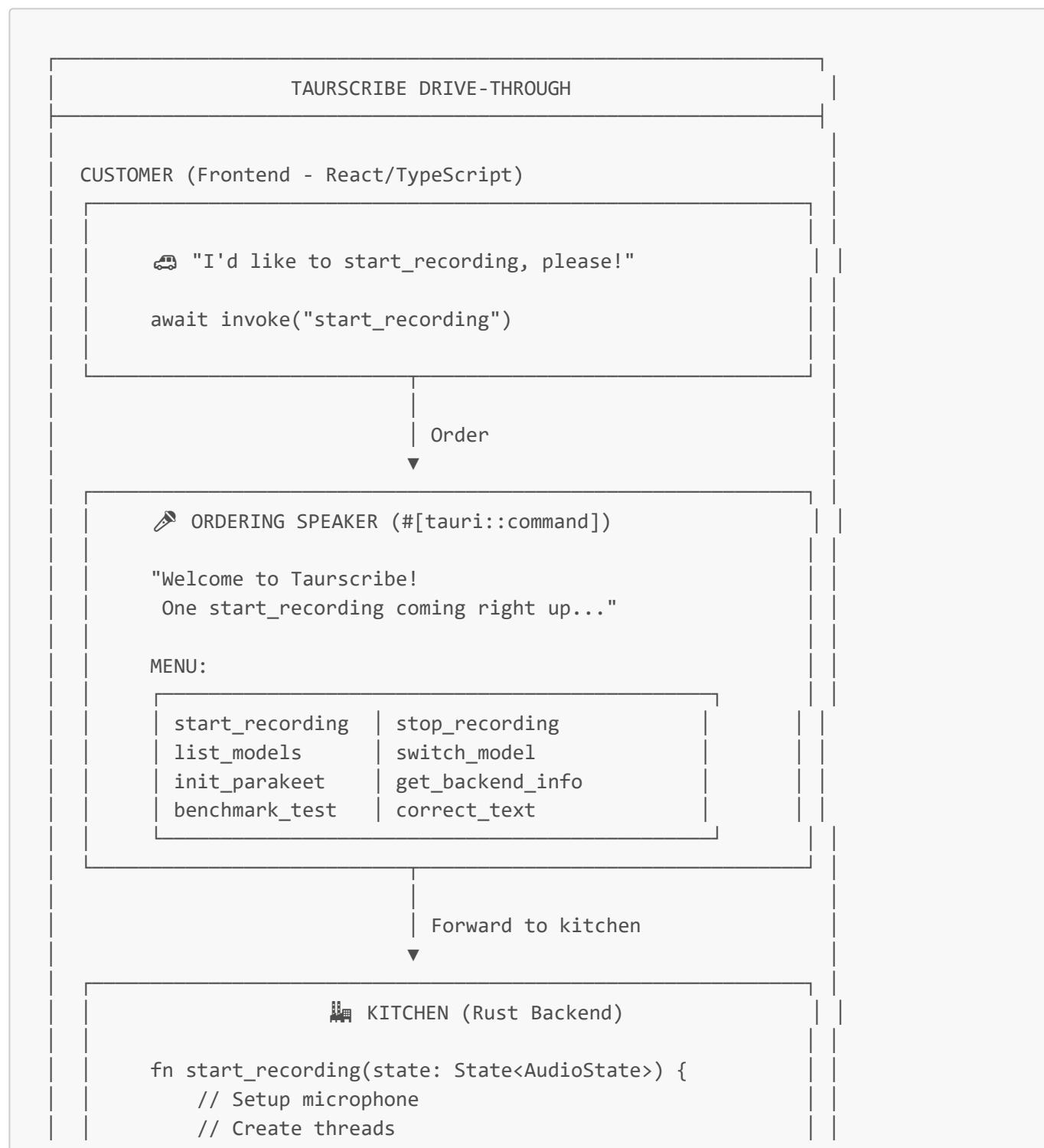
**10. Tauri Commands - The Drive-Through Window****The Code:**

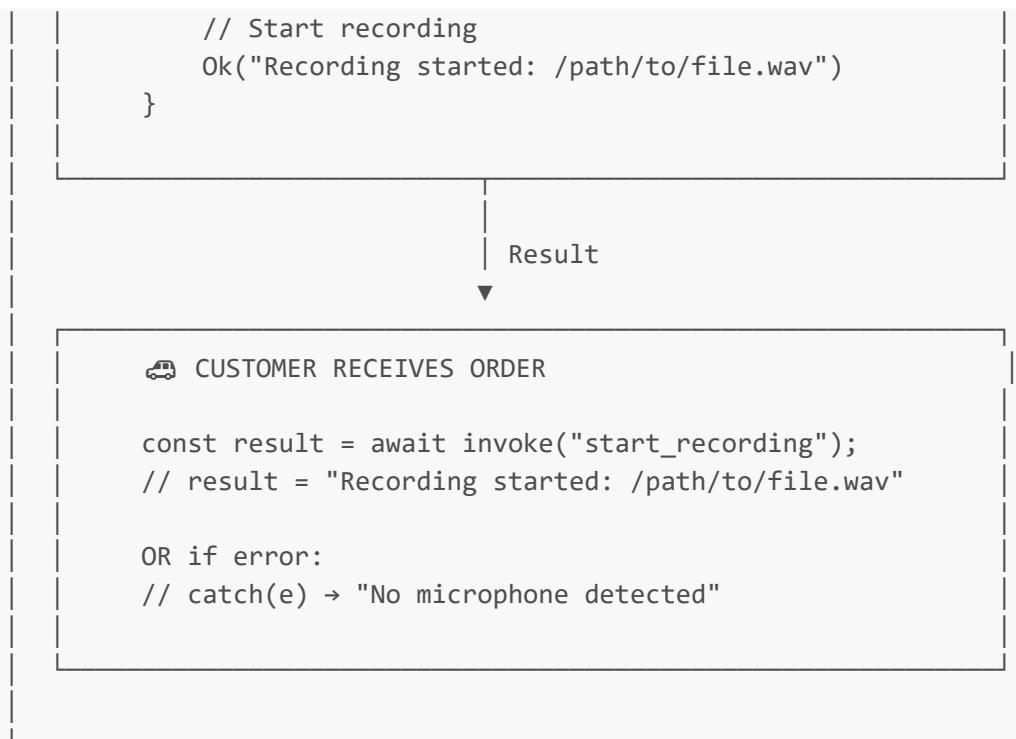
```
#[tauri::command]
pub fn start_recording(
    app_handle: AppHandle,
    state: State<AudioState>,
) -> Result<String, String> {
    // ... do stuff
    Ok("Recording started".to_string())
}
```

**The Analogy: A Fast Food Drive-Through**

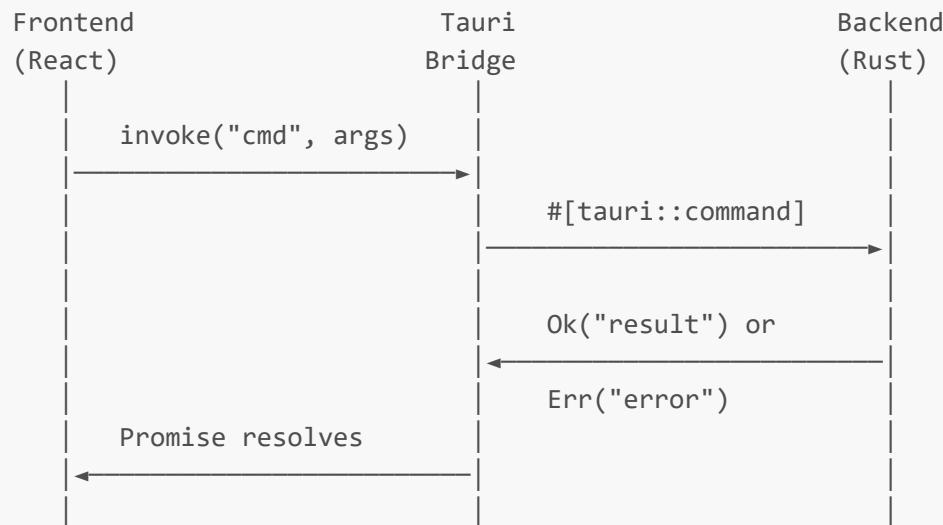
Rust Concept	Drive-Through Analogy
Frontend (React)	Customer in car

Rust Concept	Drive-Through Analogy
Tauri Command	The ordering speaker/window
<code>#[tauri::command]</code>	"This window accepts orders"
<code>State&lt;AudioState&gt;</code>	Access to the kitchen equipment
<code>AppHandle</code>	Walkie-talkie to communicate back
<code>Result&lt;String, String&gt;</code>	Either food or "Sorry, we're out"
<code>invoke("start_recording")</code>	Customer places order

**Visual:**



#### COMMUNICATION FLOW:



## Analogy Quick Reference

Complex Concept	Simple Analogy
<code>Arc&lt;Mutex&lt;T&gt;&gt;</code>	Shared notebook with a lock in a library
Channels	Sushi conveyor belt
Resampling	Currency exchange
VAD	Bouncer at a VIP club
GPU Backends	Transportation options (Ferrari → Walking)

Complex Concept	Simple Analogy
Whisper Context	Stenographer's notepad
Option<T>	Mailbox (might be empty)
Result<T, E>	Package delivery (success or failure)
unsafe Send/Sync	Liability waiver at adventure park
Thread Architecture	Restaurant kitchen with multiple cooks
Tauri Commands	Fast food drive-through window

## 🔗 Data Type Decisions: Why These Types?

This section explains the reasoning behind every major data type choice in the codebase, including why alternatives were rejected.

### 1. Arc<Mutex<T>> for Shared State

#### What we chose:

```
pub struct AudioState {
    pub whisper: Arc<Mutex<WhisperManager>>,
    pub vad: Arc<Mutex<VADManager>>,
    pub parakeet: Arc<Mutex<ParakeetManager>>,
    // ...
}
```

#### Why Arc<Mutex<T>>?

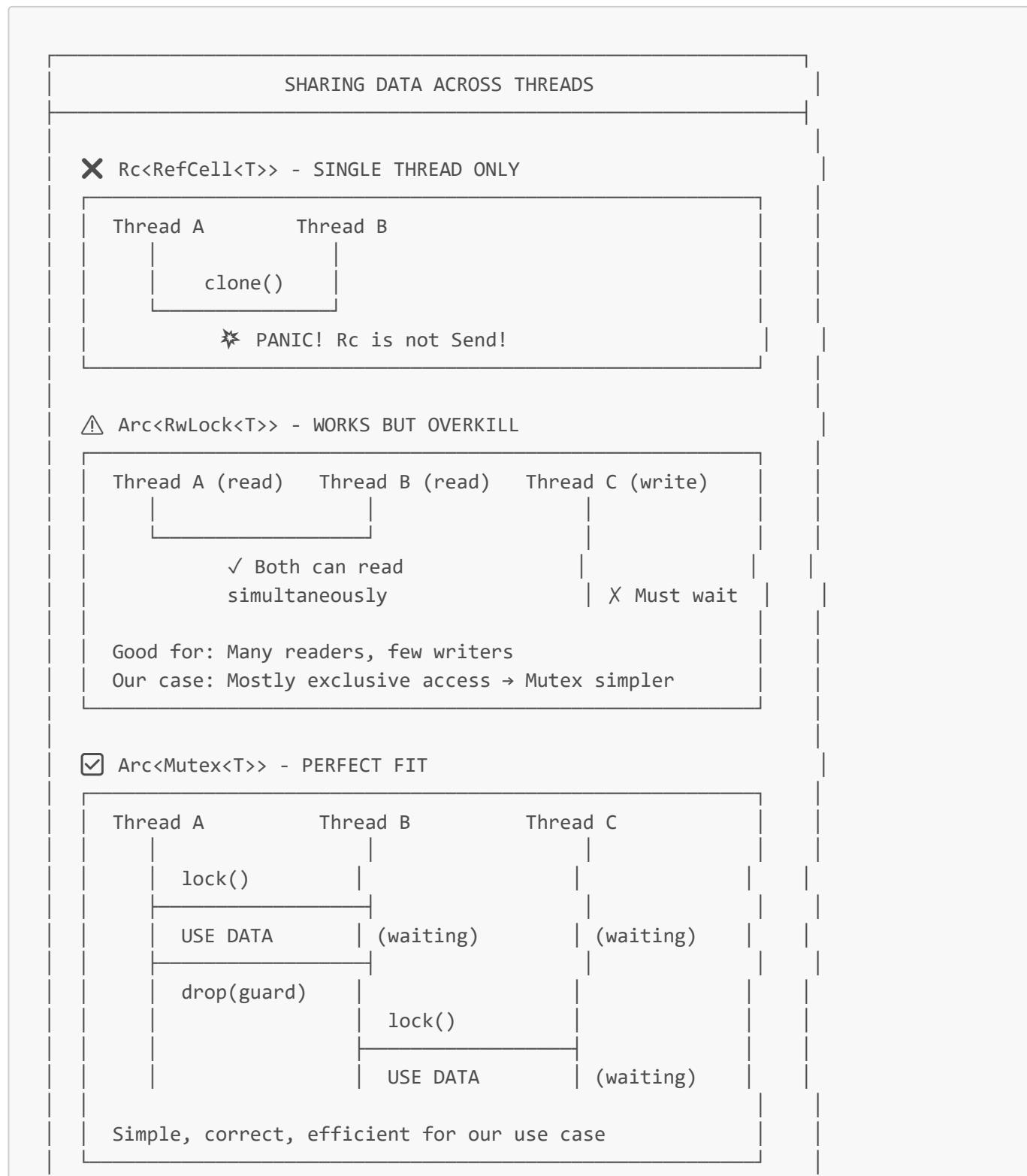
Requirement	Why Arc<Mutex> Satisfies It
Multiple threads need access	Arc = Atomic Reference Counting (thread-safe sharing)
Threads need to modify data	Mutex = Mutual Exclusion (one writer at a time)
Owned by Tauri state system	Arc allows cloning handles without moving ownership

#### Alternatives Considered:

Alternative	Why NOT Used
Rc<RefCell<T>>	✗ NOT thread-safe! Rc is single-threaded only. Would panic if used across threads.
Arc<RwLock<T>>	⚠ Could work, but overkill. RwLock allows multiple readers OR one writer. Our access patterns are mostly exclusive writes (recording), so Mutex is simpler.
Arc<T> (no Mutex)	✗ Only allows read-only access. We need to modify WhisperManager's internal state.

Alternative	Why NOT Used
Pass by argument	✗ Can't pass owned values to spawned threads that outlive the function.
Global static	✗ Rust makes global mutable state intentionally hard. Would need <code>lazy_static</code> + <code>Mutex</code> anyway.
Channels only	⚠ Could work for some cases, but we need synchronous access (lock, use, unlock) not async message passing.

### Visual comparison:



## 2. `Vec<f32>` for Audio Samples

### What we chose:

```
pub fn transcribe_chunk(&mut self, samples: &[f32], sample_rate: u32) ->
Result<String, String>
```

### Why `f32` (not `f64`, `i16`, `i32`)?

Type	Pros	Cons	Verdict
<code>f32</code>	Whisper expects f32, GPU optimized for f32, half the memory of f64	Slightly less precision	<input checked="" type="checkbox"/> CHOSEN
<code>f64</code>	More precision	Whisper doesn't need it, 2x memory, slower on GPU	<input type="checkbox"/> Overkill
<code>i16</code>	Native WAV format, compact	Need to convert to f32 anyway for Whisper	<input type="checkbox"/> Extra conversion step
<code>i32</code>	More range than i16	Still need f32 conversion, uncommon format	<input type="checkbox"/> No benefit

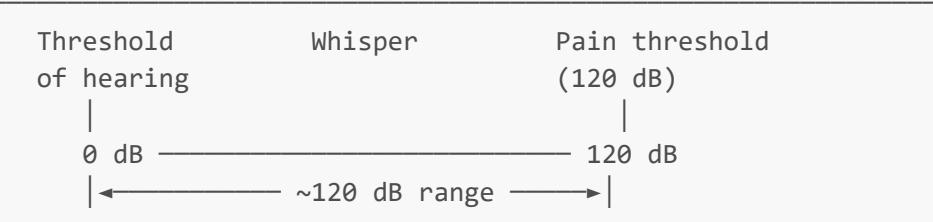
### Why `Vec<f32>` (not `&[f32]`, `Box<[f32]>`, `ArrayVec`)?

Type	Pros	Cons	Verdict
<code>Vec&lt;f32&gt;</code>	Growable, owned, can be moved between threads	Heap allocation	<input checked="" type="checkbox"/> CHOSEN
<code>&amp;[f32]</code>	No allocation, fast	Can't own, can't send to thread, lifetime issues	<input type="checkbox"/> Can't move to spawned threads
<code>Box&lt;[f32]&gt;</code>	Owned, fixed size	Can't grow, awkward to create	<input type="checkbox"/> Need growable for accumulation
<code>ArrayVec&lt;f32, N&gt;</code>	Stack allocated, no heap	Fixed max size, can't handle variable audio lengths	<input type="checkbox"/> Audio chunks vary in size

### Why `f32` precision is enough:

#### AUDIO PRECISION COMPARISON

HUMAN HEARING DYNAMIC RANGE:



#### f32 PRECISION:

- 23 bits of mantissa = ~140 dB dynamic range
- MORE than human hearing needs!
- Standard for audio processing industry

#### f64 PRECISION:

- 52 bits of mantissa = ~310 dB dynamic range
- WAY more than needed
- 2x memory, slower processing
- Used for scientific computing, not audio

CONCLUSION: f32 is perfect. f64 wastes resources.

### 3. String vs &str for Transcripts

#### What we chose:

```
pub struct WhisperManager {
    last_transcript: String, // Owned string
    // ...
}

pub fn transcribe_chunk(&mut self, ...) -> Result<String, String> // Returns
owned
```

#### Why String (not &str)?

Scenario	String	&str
Store in struct	<input checked="" type="checkbox"/> Owns the data	<input checked="" type="checkbox"/> Needs lifetime parameter, can't outlive source
Return from function	<input checked="" type="checkbox"/> Caller owns result	<input checked="" type="checkbox"/> What does it borrow from? Lifetime hell
Modify (append)	<input checked="" type="checkbox"/> .push_str() works	<input checked="" type="checkbox"/> Immutable by nature
Send to frontend	<input checked="" type="checkbox"/> Serializes cleanly	<input checked="" type="checkbox"/> Works but must clone anyway

Scenario	String	&str
Accumulate context	<input checked="" type="checkbox"/> Can grow	✗ Fixed size slice

### The lifetime problem with &str:

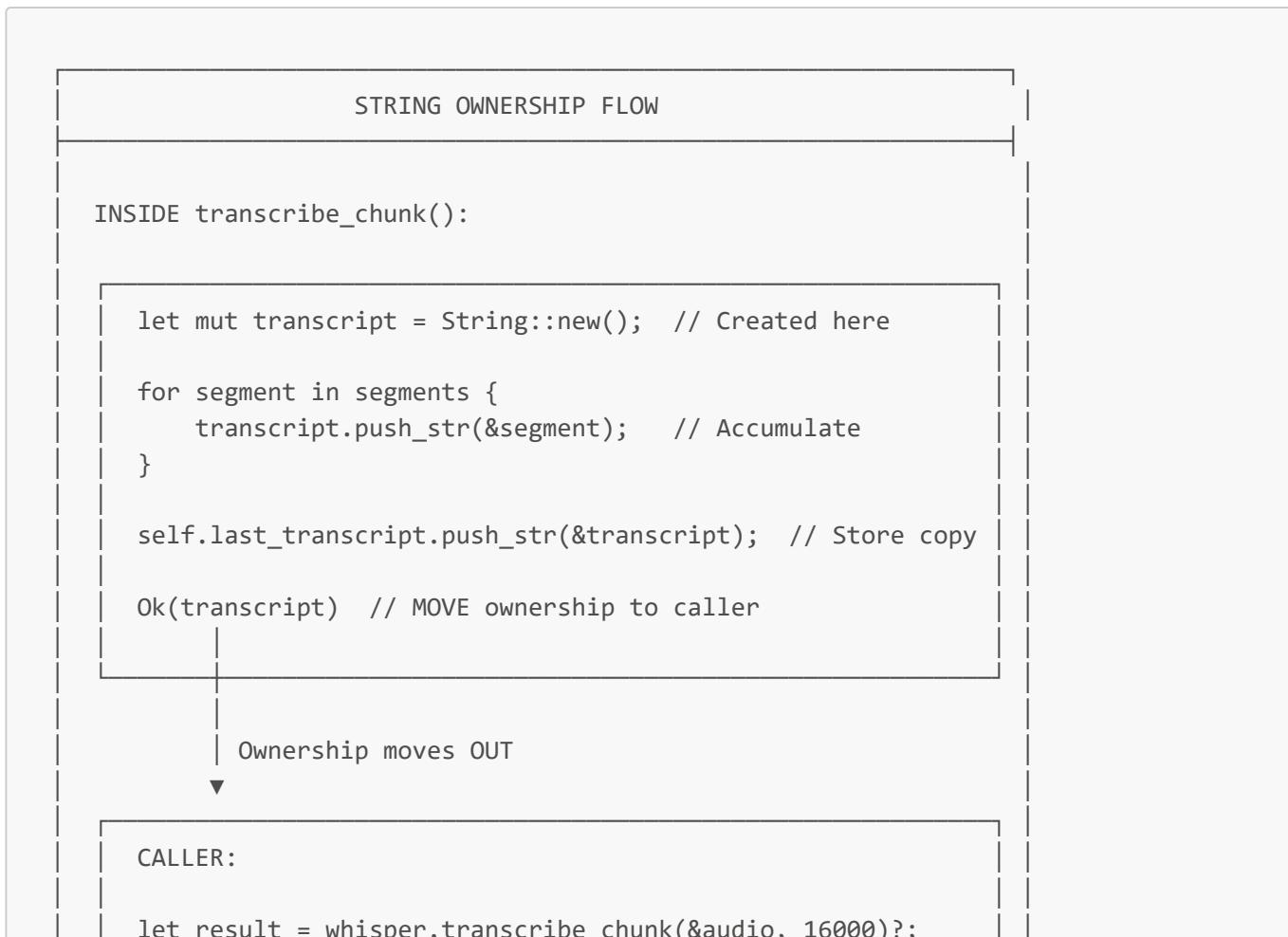
```
// ✗ THIS WON'T COMPILE
pub struct WhisperManager {
    last_transcript: &str, // Error: missing lifetime specifier
}

// ✗ EVEN WITH LIFETIME, STILL BROKEN
pub struct WhisperManager<'a> {
    last_transcript: &'a str, // But where does 'a come from?
}

// The string is CREATED inside transcribe_chunk(), so it can't
// outlive the function. We'd be returning a reference to a local!

// ☑ THIS WORKS
pub struct WhisperManager {
    last_transcript: String, // Owns the data, no lifetime issues
}
```

### Visual:



```
//      ^^^^^^ Now OWNS the String
app.emit("transcription", result); // Can move again
```

If we used &str:

```
let transcript = String::new(); // Created here
// ...
Ok(&transcript) // ✗ ERROR! Can't return reference to
                // local variable - it's about to die!
```

## 4. Option<T> vs Sentinel Values

### What we chose:

```
pub struct WhisperManager {
    context: Option<WhisperContext>, // None = not loaded
    current_model: Option<String>, // None = no model selected
}
```

### Why Option<T> (not sentinel values)?

Approach	Example	Pros	Cons
Option<T>	context: Option<WhisperContext>	Type-safe, compiler enforces handling	Extra enum wrapper
Sentinel value	model_name: String with "" meaning "none"	No wrapper	✗ Easy to forget check, "" might be valid
Nullable pointer	context: *mut WhisperContext with null	C-compatible	✗ Unsafe, no compiler help
Boolean flag	is_loaded: bool + context: WhisperContext	Explicit	✗ Can get out of sync, can't have uninitialized context

### The sentinel value trap:

```
// ✗ BAD: Using empty string as "none"
pub struct WhisperManager {
    current_model: String, // "" means no model
}

// Easy to forget the check:
fn get_model_name(&self) -> &str {
```

```

    &self.current_model // Oops! Returns "" instead of indicating "none"
}

// What if "" is a valid model name? (unlikely but possible)
// The compiler can't help you remember to check!

//  GOOD: Using Option
pub struct WhisperManager {
    current_model: Option<String>,
}

// Compiler FORCES you to handle both cases:
fn get_model_name(&self) -> Option<&String> {
    self.current_model.as_ref() // Explicit: might be None
}

// Can't accidentally use a None value:
let name = whisper.current_model; // Type is Option<String>, not String
// Must handle: name.unwrap(), name.expect(), match name { ... }, etc.

```

**Visual:**

## OPTION&lt;T&gt; vs SENTINEL VALUES

## SENTINEL VALUE (Bad):

```

current_model: String

Value: "tiny.en"      → Model is tiny.en
Value: "large-v3"     → Model is large-v3
Value: ""              → No model? Or bug? Or valid?
^ ^
⚠️ Ambiguous! Is "" intentional or a mistake?

COMPILER: "Looks fine to me! 🤖"

```

## OPTION&lt;T&gt; (Good):

```

current_model: Option<String>

Value: Some("tiny.en")  → Model is tiny.en
Value: Some("large-v3") → Model is large-v3
Value: None             → EXPLICITLY no model
^ ^ ^
 Crystal clear! Impossible to misinterpret.

```

COMPILER: "You MUST handle the None case! ⚡"

## 5. Result<T, String> vs Custom Error Types

### What we chose:

```
pub fn transcribe_chunk(&mut self, ...) -> Result<String, String>
pub fn initialize(&mut self, ...) -> Result<String, String>
```

### Why Result<T, String> (not custom error enum)?

Approach	Example	Pros	Cons
Result<T, String>	Result<String, String>	Simple, human-readable errors, easy to serialize to frontend	Less structured, can't match on error types
Custom error enum	Result<String, WhisperError>	Pattern matching, structured	More boilerplate, overkill for Tauri commands
anyhow::Error	Result<String, anyhow::Error>	Ergonomic, chains errors	External dependency, less control
thiserror	Custom error with derive macro	Best of both worlds	More code, more complexity

### Why String is fine for this project:

```
// Our errors go directly to the frontend:
#[tauri::command]
pub fn start_recording(...) -> Result<String, String> {
    // Error needs to be displayable to user anyway
    let device = host.default_input_device()
        .ok_or("No microphone detected")?; // User-friendly message

    // ...
}

// In frontend (TypeScript):
try {
    await invoke("start_recording");
} catch (error) {
    showToast(error); // Error is already a nice string!
}
```

## When you WOULD use custom errors:

```
// If we needed to handle errors differently based on type:
enum WhisperError {
    ModelNotFound(String),
    GpuUnavailable,
    AudioTooShort,
    NetworkError(std::io::Error),
}

// Then we could do:
match whisper.transcribe_chunk(&audio, 16000) {
    Ok(text) => show_text(text),
    Err(WhisperError::GpuUnavailable) => fallback_to_cpu(), // Specific handling
    Err(WhisperError::AudioTooShort) => wait_for_more_audio(),
    Err(e) => show_error(e),
}

// But our app just shows errors to user, so String is simpler.
```

## 6. crossbeam_channel vs std::sync::mpsc

### What we chose:

```
use crossbeam_channel::unbounded;

let (file_tx, file_rx) = unbounded::<Vec<f32>>();
```

### Why crossbeam_channel (not std::sync::mpsc)?

Feature	std::sync::mpsc	crossbeam_channel
Multi-producer	<input checked="" type="checkbox"/> Yes (mpsc)	<input checked="" type="checkbox"/> Yes
Multi-consumer	<input type="checkbox"/> No (single consumer)	<input checked="" type="checkbox"/> Yes (mpmc)
Clone receiver	<input type="checkbox"/> Can't clone Receiver	<input checked="" type="checkbox"/> Can clone
Select/try_recv	<input type="checkbox"/> Limited	<input checked="" type="checkbox"/> Full support
Performance	Good	Better (lock-free)
Bounded channels	<input type="checkbox"/> sync_channel only	<input checked="" type="checkbox"/> First-class

### Our specific need:

```
// We clone the sender to give to multiple producers:
let file_tx_clone = file_tx.clone(); // For audio callback
let whisper_tx_clone = whisper_tx.clone(); // For audio callback
```

```
// std::sync::mpsc can do this (clone sender is fine)  
// BUT crossbeam is faster and we might need multi-consumer later
```

### Visual comparison:

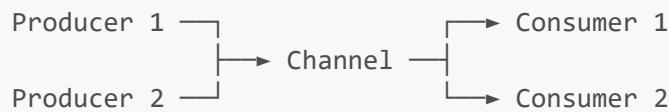
#### CHANNEL COMPARISON

`std::sync::mpsc (Multi-Producer, Single-Consumer):`



✗ Can't have: Channel —————→ Consumer 1  
                                  └————→ Consumer 2

`crossbeam_channel (Multi-Producer, Multi-Consumer):`



- Can clone both sender AND receiver
- Lock-free implementation (faster)
- Better select! macro support

#### WHY WE CHOSE CROSSBEAM:

1. Already a dependency (via other crates)
2. Better performance in high-throughput audio scenario
3. Flexibility if we need multi-consumer later
4. Cleaner API for bounded channels

## 7. **u32** for Sample Rate vs Other Integer Types

### What we chose:

```
pub fn transcribe_chunk(&mut self, samples: &[f32], input_sample_rate: u32) ->
Result<String, String>
```

### Why u32 (not usize, u16, i32)?

Type	Range	Verdict
u32	0 to 4,294,967,295	<input checked="" type="checkbox"/> Matches audio industry standard, can represent any sample rate
usize	Platform-dependent (32 or 64 bit)	<input checked="" type="checkbox"/> Overkill, inconsistent across platforms
u16	0 to 65,535	<input checked="" type="checkbox"/> Too small! 192kHz audio wouldn't fit
i32	-2B to +2B	<input checked="" type="checkbox"/> Negative sample rates make no sense
u64	0 to 18 quintillion	<input checked="" type="checkbox"/> Massive overkill, wastes memory

### Real-world sample rates:

COMMON SAMPLE RATES		
Sample Rate	Use Case	Fits in u32?
8,000 Hz	Telephone	<input checked="" type="checkbox"/> Yes
16,000 Hz	Whisper AI input	<input checked="" type="checkbox"/> Yes
22,050 Hz	AM Radio	<input checked="" type="checkbox"/> Yes
44,100 Hz	CD Audio	<input checked="" type="checkbox"/> Yes
48,000 Hz	Standard microphone	<input checked="" type="checkbox"/> Yes
96,000 Hz	High-res audio	<input checked="" type="checkbox"/> Yes
192,000 Hz	Studio master	<input checked="" type="checkbox"/> Yes
384,000 Hz	Extreme audiophile	<input checked="" type="checkbox"/> Yes

u32 max: 4,294,967,295 Hz = 4.3 GHz  
That's 4.3 BILLION samples per second!  
No audio needs anywhere near this.

u16 max: 65,535 Hz  
 Would NOT fit 96kHz or 192kHz!

## 8. PathBuf vs String for File Paths

### What we chose:

```
fn get_models_dir() -> Result<std::path::PathBuf, String>
let absolute_path = models_dir.join(&file_name);
```

## Why PathBuf (not String)?

Feature	String	PathBuf
Cross-platform	✗ / vs \ issues	<input checked="" type="checkbox"/> Handles automatically
Path manipulation	✗ Manual string concat	<input checked="" type="checkbox"/> .join(), .parent(), .extension()
Unicode handling	⚠ Assumes UTF-8	<input checked="" type="checkbox"/> Handles OS-native encoding
Type safety	✗ Any string is "valid"	<input checked="" type="checkbox"/> Clearly indicates "this is a path"

## The cross-platform problem:

```
// ✗ BAD: Using String for paths
let models_dir = "taurscribe-runtime/models".to_string();
let model_path = models_dir + "/" + &file_name; // Only works on Unix!
// On Windows: "taurscribe-runtime/models/ggml-tiny.bin"
//               (forward slash might work, but inconsistent)

// ☑ GOOD: Using PathBuf
let models_dir = PathBuf::from("taurscribe-runtime/models");
let model_path = models_dir.join(&file_name); // Works everywhere!
// On Windows: "taurscribe-runtime\models\ggml-tiny.bin"
// On Unix:     "taurscribe-runtime/models/ggml-tiny.bin"
```

## PathBuf superpowers:

```
let path = PathBuf::from("/home/user/models/ggml-tiny.en-q5_1.bin");

path.file_name()      // Some("ggml-tiny.en-q5_1.bin")
path.extension()     // Some("bin")
path.parent()         // Some("/home/user/models")
path.is_absolute()   // true
path.exists()         // true/false (filesystem check)
path.join("subdir")  // "/home/user/models/ggml-tiny.en-q5_1.bin/subdir"

// With String, you'd have to implement all of this manually!
```

---

## 9. Enum vs Struct for State Machines

### What we chose:

```
// For GPU backend (fixed set of options):
pub enum GpuBackend {
    Cuda,
    Vulkan,
    Cpu,
}

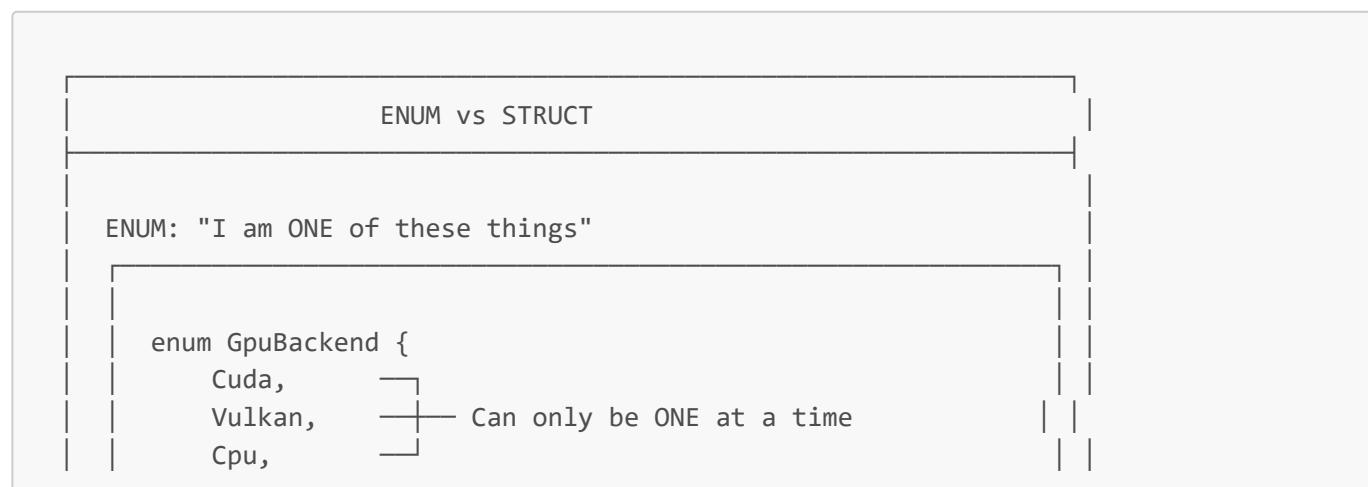
// For app state (fixed set of states):
pub enum AppState {
    Ready,
    Recording,
    Processing,
}

// For model info (multiple fields, not alternatives):
pub struct ModelInfo {
    pub id: String,
    pub display_name: String,
    pub file_name: String,
    pub size_mb: f32,
}
```

### When to use Enum vs Struct:

Use Case	Enum	Struct
<b>One of many options</b>	<input checked="" type="checkbox"/> GpuBackend::Cuda	✗
<b>Multiple fields together</b>	✗	<input checked="" type="checkbox"/> ModelInfo { id, name, ... }
<b>State machine</b>	<input checked="" type="checkbox"/> AppState::Recording	✗
<b>Data container</b>	✗	<input checked="" type="checkbox"/> TranscriptionChunk { text, time }
<b>Pattern matching</b>	<input checked="" type="checkbox"/> match backend { ... }	⚠ Possible but awkward
<b>Can be in multiple states</b>	✗	<input checked="" type="checkbox"/> (use multiple bools/fields)

### Visual:



```
}
```

```
let backend = GpuBackend::Cuda;
// It's Cuda. Not Vulkan. Not Cpu. Just Cuda.
```

STRUCT: "I have ALL of these things"

```
struct ModelInfo {
    id: String,
    display_name: String, ┌─────────┐ Has ALL of these
    file_name: String,   ┌─────────┐ simultaneously
    size_mb: f32,
}

let model = ModelInfo {
    id: "tiny.en".into(),
    display_name: "Tiny English".into(),
    file_name: "ggml-tiny.en.bin".into(),
    size_mb: 42.3,
};
// Has ALL four fields at once
```

WRONG CHOICES:

```
// ❌ BAD: Struct for mutually exclusive options
struct GpuBackend {
    is_cuda: bool,
    is_vulkan: bool,
    is_cpu: bool,
}
// Problem: What if is_cuda AND is_vulkan are both true?
// Enum makes this impossible by design!

// ❌ BAD: Enum for independent properties
enum ModelInfo {
    Id(String),
    DisplayName(String),
    FileName(String),
    SizeMb(f32),
}
// Problem: A model has ALL of these, not just one!
```

## 10. Tuple vs Named Struct for Resampler Cache

### What we chose:

```
pub struct WhisperManager {
    resampler: Option<(u32, usize, Box<SincFixedIn<f32>>)>,
    //           ^^^^ ^^^^^^ ^^^^^^^^^^^^^^^^^^^^^^
    //           |   |   | The actual resampler
    //           |   |   | Expected input chunk size
    //           |   |   |
    //           |   |   | Source sample rate
}
```

### Why a tuple here (not a named struct)?

Approach	Pros	Cons
Tuple (u32, usize, Box<...>)	Concise, private implementation detail	Less readable at call site
Named struct	Self-documenting field names	More boilerplate for internal detail

### When tuples are okay:

```
// ✅ GOOD: Tuple for private, simple grouping
resampler: Option<(u32, usize, Box<SincFixedIn<f32>>)>

// Usage is internal only:
if let Some((rate, size, resampler)) = &mut self.resampler {
    // Destructure gives names anyway
}

// ⚠️ WOULD USE STRUCT IF:
// - It was part of the public API
// - It had more than 3-4 fields
// - The meaning wasn't obvious from context
```

### When to definitely use a named struct:

```
// ❌ BAD: Tuple with too many elements
let config: (u32, u32, f64, bool, String, usize) = ...;
// What is each field?! 🤔

// ✅ GOOD: Named struct for clarity
struct ResamplerConfig {
    source_rate: u32,
    target_rate: u32,
    quality: f64,
```

```

    preserve_length: bool,
    algorithm: String,
    chunk_size: usize,
}

```

## 11. `Box<T>` for Large Types

### What we chose:

```

resampler: Option<(u32, usize, Box<SincFixedIn<f32>>)>
//                                     ^^^^
//                                         Boxed!

```

### Why `Box<SincFixedIn<f32>>` (not just `SincFixedIn<f32>`)?

Storage	<code>SincFixedIn&lt;f32&gt;</code>	<code>Box&lt;SincFixedIn&lt;f32&gt;&gt;</code>
Location	Stack (inline in struct)	Heap (pointer in struct)
Size	Large (internal buffers)	8 bytes (just a pointer)
Move cost	Expensive (copy all data)	Cheap (copy pointer)

### The problem with large stack types:

#### STACK vs HEAP STORAGE

WITHOUT `Box` (everything on stack):

```

WhisperManager {
    context: Option<...>,           // ~100 bytes
    last_transcript: String,        // 24 bytes
    backend: GpuBackend,            // 1 byte
    current_model: Option<...>,    // 24 bytes
    resampler: Option<(
        u32,                      // 4 bytes
        usize,                     // 8 bytes
        SincFixedIn<f32>,         // ~50,000 bytes!! 🎈
    )>,                         // (internal buffers)
}

Total: ~50KB on stack! Stack overflow risk!

```

WITH `Box` (large data on heap):

```

WhisperManager (on stack) {
    context: Option<...>,           // ~100 bytes
    last_transcript: String,        // 24 bytes
    backend: GpuBackend,            // 1 byte
    current_model: Option<...>,    // 24 bytes
    resampler: Option<(
        u32,                      // 4 bytes
        usize,                     // 8 bytes
        Box<SincFixedIn<f32>>,   // 8 bytes (pointer)
    )>,
}

```

Total stack: ~170 bytes

### HEAP

```

SincFixedIn<f32> {
    internal_buffers: [...50KB of data...]
}

```

Heap is much larger than stack - no problem!

## Quick Reference: Type Decision Tree

### WHICH TYPE SHOULD I USE?

#### SHARING DATA BETWEEN THREADS?

- └ Yes, need write access → Arc<Mutex<T>>
- └ Yes, mostly read, rare write → Arc<RwLock<T>>
- └ No, single thread → Rc<RefCell<T>> or just T

#### RETURNING VALUE FROM FUNCTION?

- └ Might not exist → Option<T>
- └ Might fail → Result<T, E>
- └ Always exists → T

#### STORING TEXT?

- └ Need to own/modify → String
- └ Just reading, no ownership → &str

#### COLLECTION OF ITEMS?

```

    └─ Unknown size, growable → Vec<T>
    └─ Fixed size at compile time → [T; N]
    └─ Key-value lookup → HashMap<K, V>

FILE PATH?
└─ Always use PathBuf (not String)

MUTUALLY EXCLUSIVE OPTIONS?
└─ Use Enum

MULTIPLE PROPERTIES TOGETHER?
└─ Use Struct

LARGE DATA STRUCTURE?
└─ Box<T> to put it on heap

THREAD COMMUNICATION?
└─ Async message passing → crossbeam_channel
└─ Sync shared access → Arc<Mutex<T>>

NUMERIC TYPE?
└─ Audio samples → f32
└─ Array index → usize
└─ Sample rate → u32
└─ Can be negative → i32/i64

```

## 🦀 Rust Fundamentals for Beginners

This section teaches core Rust concepts using real examples from this codebase. If you're new to Rust, read this before diving into the code!

### 1. Ownership: Rust's Superpower

**The Golden Rule:** Every value in Rust has exactly ONE owner. When the owner goes out of scope, the value is dropped (freed).

**From our codebase:**

```

// In recording.rs
let filename = format!("recording_{}.wav", chrono::Utc::now().timestamp());
// ^^^^^^
// `filename` OWNS this String

let path = recordings_dir.join(&filename);
// ^^^
// BORROWING filename (not taking ownership)

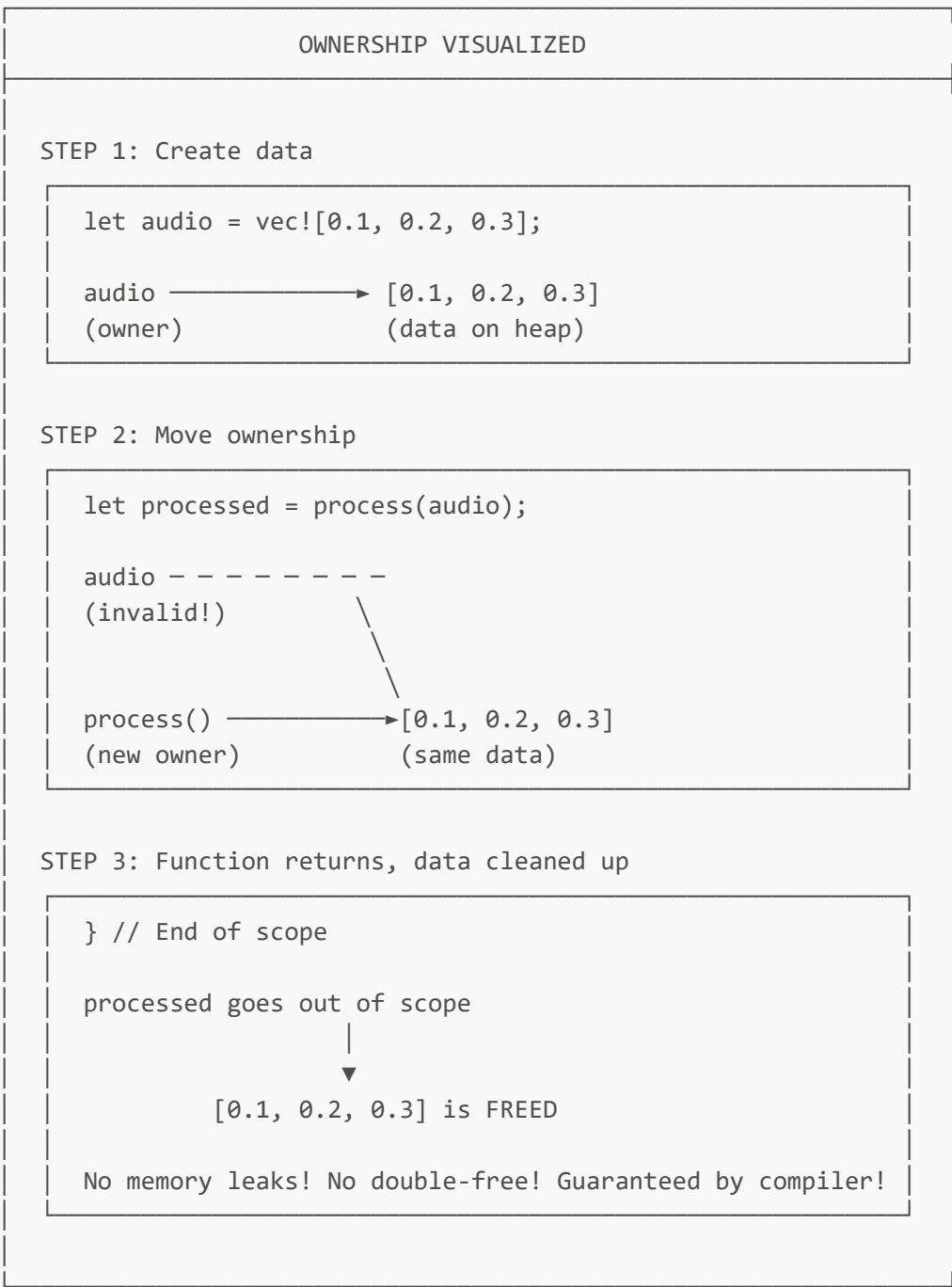
```

```
println!("Saving to: {}", filename); // Still works! We only borrowed above
```

## What happens when ownership moves:

```
// EXAMPLE: Ownership transfer
let audio = vec![0.1, 0.2, 0.3]; // `audio` owns the Vec
let processed = process(audio); // Ownership MOVES to `process()`
// println!("{:?}", audio); // ✗ ERROR! `audio` no longer owns the data
```

### Visual:



## 2. Borrowing: Using Without Owning

### Two types of borrows:

- `&T` = Immutable borrow (can read, can't modify, many allowed)
- `&mut T` = Mutable borrow (can modify, only ONE at a time)

### From our codebase:

```
// In whisper.rs - Immutable borrow
pub fn transcribe_chunk(
    &mut self,           // Mutable borrow of self (we'll modify internal state)
    samples: &[f32],     // Immutable borrow of samples (just reading)
    input_sample_rate: u32,
) -> Result<String, String> {
    // We can READ samples but not modify them
    let sum: f32 = samples.iter().sum(); // ✓ Reading is fine
    // samples[0] = 0.0;                // ✗ ERROR! Can't modify

    // We CAN modify self because we have &mut self
    self.last_transcript.push_str("text"); // ✓ Modifying is fine
}
```

### The borrowing rules:

#### BORROWING RULES

##### RULE 1: You can have EITHER:

- One mutable reference (`&mut T`)
- OR any number of immutable references (`&T`)
- BUT NOT BOTH at the same time!

```
let mut data = vec![1, 2, 3];

// ✓ VALID: Multiple immutable borrows
let r1 = &data;
let r2 = &data;
println!("{} {}", r1, r2); // Both reading - OK!

// ✓ VALID: Single mutable borrow
let r3 = &mut data;
r3.push(4); // Modifying - OK!

// ✗ INVALID: Mutable + immutable at same time
let r4 = &data; // Immutable borrow
let r5 = &mut data; // ERROR! Can't borrow mutably
```

```
// while immutably borrowed!
```

### WHY THIS RULE?

Prevents data races at compile time!

Thread A: reading data    }  
 Thread B: modifying data    } = DATA RACE! ✨

Rust: "You can't even COMPILE this code."

## 3. The ? Operator: Error Handling Magic

**What it does:** If the Result is `Err`, return early. If `Ok`, unwrap the value.

**From our codebase:**

```
// In whisper.rs
pub fn transcribe_file(&mut self, file_path: &str) -> Result<String, String> {
    // Without ? operator (verbose):
    let reader = match hound::WavReader::open(file_path) {
        Ok(r) => r,
        Err(e) => return Err(format!("Failed to open: {}", e)),
    };

    // With ? operator (concise):
    let reader = hound::WavReader::open(file_path)
        .map_err(|e| format!("Failed to open: {}", e))?;
    // ^^^
    // This does the same thing!

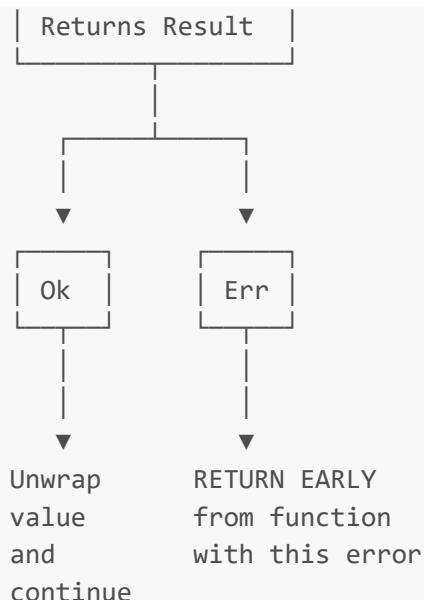
    // ... rest of function
    Ok(transcript)
}
```

**Visual:**

### THE ? OPERATOR

```
some_function()?
```





EXAMPLE:

```

fn process() -> Result<String, String> {
    let data = read_file()?;
    let parsed = parse(data)?;
    let result = compute(parsed)?;
    Ok(result)
}

// Equivalent without ?:
fn process() -> Result<String, String> {
    let data = match read_file() {
        Ok(d) => d,
        Err(e) => return Err(e),
    };
    let parsed = match parse(data) {
        Ok(p) => p,
        Err(e) => return Err(e),
    };
    // ... you get the idea, MUCH more verbose!
}
  
```

## 4. Closures: Anonymous Functions

**Syntax:** |parameters| body or |parameters| { body }

**From our codebase:**

```

// In recording.rs - Simple closure
let mono_data: Vec<f32> = data.chunks(channels)
  
```

```
.map(|chunk| chunk.iter().sum::<f32>() / channels as f32)
//      ^^^^^^ This is a closure!
//      |chunk| = takes one parameter called `chunk`
//      chunk.iter().sum::<f32>() / channels as f32 = returns this value
.collect();

// In recording.rs - Closure with move
std::thread::spawn(move || {
//           ^
//           `move` means: take ownership of captured variables
let mut writer = writer; // `writer` was moved into this closure
while let Ok(samples) = file_rx.recv() {
    // ...
}
});
```

## Understanding move:

### CLOSURES AND MOVE

WITHOUT move (borrows by default):

```
let name = String::from("Alice");

let greet = || println!("Hello, {}", name);
           ^
           Closure BORROWS `name`

greet(); // Works!
println!("{}", name); // Still works! name still valid
```

WITH move (takes ownership):

```
let name = String::from("Alice");

let greet = move || println!("Hello, {}", name);
           ^
           Closure TAKES OWNERSHIP of `name`

greet(); // Works!
// println!("{}", name); // X ERROR! name was moved
```

WHY USE move WITH THREADS?

```
std::thread::spawn(move || {
    // This closure might outlive the current function!
    // If we only borrowed, the data could be freed
    // while the thread is still using it!
```

```
// move ensures the thread OWNS the data,
// so it will live as long as the thread needs it.
});
```

## 5. Pattern Matching: The Swiss Army Knife

**match** is like **switch** on steroids.

From our codebase:

```
// In parakeet.rs - Matching on enum
match model {
    LoadedModel::Nemotron(m) => {
        // m is the inner Nemotron value
        m.transcribe_chunk(&chunk)
    }
    LoadedModel::Ctc(m) => {
        m.transcribe_samples(audio, 16000, 1, None)
    }
    LoadedModel::Eou(m) => {
        m.transcribe(&chunk, false)
    }
    LoadedModel::Tdt(m) => {
        m.transcribe_samples(audio, 16000, 1, None)
    }
}

// In whisper.rs - Matching on Result
match WhisperContext::new_with_params(path, params) {
    Ok(ctx) => {
        // Success! ctx is the WhisperContext
        Ok((ctx, GpuBackend::Cuda))
    }
    Err(e) => {
        // Failure! e is the error
        Err(format!("Failed: {:?}", e))
    }
}
```

**if let** and **while let** - Shorthand for single-pattern matching:

```
// In recording.rs
// Instead of:
match handle.take() {
    Some(recording) => { /* use recording */ }
```

```

None => { /* do nothing */ }

}

// You can write:
if let Some(recording) = handle.take() {
    // Only runs if it's Some
    // `recording` is the inner value
}

// while let - Loop while pattern matches
while let Ok(samples) = file_rx.recv() {
    // Loops as long as recv() returns Ok
    // Stops when recv() returns Err (channel closed)
}

```

**Visual:**

## PATTERN MATCHING

```

match value {
    Pattern1 => action1,
    Pattern2 => action2,
    _ => default_action, // _ matches anything
}

```

## PATTERNS CAN:

```

// Destructure enums
Some(value) => use(value),
None => handle_none(),

// Destructure tuples
(x, y, z) => compute(x, y, z),

// Destructure structs
Point { x, y } => draw(x, y),

// Match ranges
0..=9 => println!("single digit"),
10..=99 => println!("double digit"),

// Match with guards
n if n > 0 => println!("positive"),
n if n < 0 => println!("negative"),
_ => println!("zero"),

// Match multiple patterns
Key::Left | Key::Right => move_horizontal(),

```

MUST BE EXHAUSTIVE:

```
enum Color { Red, Green, Blue }

match color {
    Color::Red => "red",
    Color::Green => "green",
    // ✗ ERROR! Missing Color::Blue case!
}

// Compiler: "You MUST handle all cases!"
// This prevents bugs when you add new enum variants.
```

## 6. Iterators and Method Chaining

**Iterators are lazy** - they don't do anything until consumed.

**From our codebase:**

```
// In whisper.rs - Complex iterator chain
let mono_samples = if spec.channels == 2 {
    samples
        .chunks(2)                                // Split into pairs [L,R]
        .map(|chunk| (chunk[0] + chunk[1]) / 2.0) // Average each pair
        .collect::<Vec<f32>>()                // Collect into Vec
} else {
    samples
};

// In vad.rs - RMS calculation
let sum_squares: f32 = audio
    .iter()                               // Create iterator over &f32
    .map(|&x| x * x)                     // Square each sample
    .sum();                             // Sum them all

// In whisper.rs - Finding models
let entries = std::fs::read_dir(&models_dir)?
    .filter_map(|e| e.ok())    // Keep only Ok values
    .filter(|e| e.path().is_file()) // Keep only files
    .filter(|e| {
        e.file_name()
            .to_str()
            .map(|n| n.starts_with("ggml-"))
```

```
.unwrap_or(false)
});
```

## Common iterator methods:

### ITERATOR CHEAT SHEET

#### CREATING ITERATORS:

```
vec.iter()          // Iterator of &T (borrowing)
vec.iter_mut()     // Iterator of &mut T (mutable borrow)
vec.into_iter()    // Iterator of T (takes ownership)
(0..10)           // Range iterator
```

#### TRANSFORMING:

```
.map(|x| x * 2)      // Transform each element
.filter(|x| x > 5)   // Keep elements matching condition
.filter_map(|x| ...) // Map + filter out None values
.flat_map(|x| ...)   // Map then flatten nested iterators
.take(5)              // Take first 5 elements
.skip(3)              // Skip first 3 elements
.enumerate()          // Add index: (0, val), (1, val)...
.zip(other)            // Pair with another iterator
.chain(other)          // Concatenate iterators
.flatten()             // Flatten nested iterators
```

#### CONSUMING (these actually run the iterator):

```
.collect::<Vec<_>>() // Collect into a collection
.sum()                  // Sum all elements
.product()              // Multiply all elements
.count()                // Count elements
.find(|x| ...)         // Find first matching element
.any(|x| ...)           // True if any element matches
.all(|x| ...)           // True if all elements match
.for_each(|x| ...)       // Run side effect on each
.fold(init, |acc, x| ...) // Reduce to single value
```

#### EXAMPLE CHAIN:

```
let result = (0..100)           // 0, 1, 2, ..., 99
    .filter(|n| n % 2 == 0)    // 0, 2, 4, ..., 98
    .map(|n| n * n)           // 0, 4, 16, ..., 9604
    .take(5)                  // 0, 4, 16, 36, 64
    .sum::<i32>();            // 120
```

## 7. Derive Macros: Auto-Implementing Traits

**#[derive(...)] automatically implements traits for your types.**

**From our codebase:**

```
// In types.rs
#[derive(Debug, Clone, Copy, PartialEq, serde::Serialize, serde::Deserialize)]
pub enum ASREngine {
    Whisper,
    Parakeet,
}
```

**What each derive does:**

COMMON DERIVES	
DERIVE	WHAT IT ENABLES
Debug	<code>println!("{}:?}", value)</code> // Debug formatting Prints struct/enum internals
Clone	<code>value.clone()</code> // Deep copy Creates a duplicate of the value
Copy	Implicit copy on assignment <code>let x = y;</code> // x is a copy, y still valid Only for small, stack-only types
PartialEq	<code>value1 == value2</code> // Equality comparison Enables == and != operators
Eq	Marker that equality is total (NaN != NaN, so f32 can't be Eq)
PartialOrd	<code>value1 &lt; value2</code> // Ordering Enables <, >, <=, >= operators
Ord	Marker that ordering is total Enables sorting
Hash	Can be used as HashMap key Enables use in HashSet

Default	Type::default() // Default value Creates a "zero" or "empty" instance
serde::Serialize	Convert to JSON/bytes Enables sending to frontend via Tauri
serde::Deserialize	Convert from JSON/bytes Enables receiving from frontend

## WHICH TO USE WHEN:

- Almost always: #[derive(Debug)]  
(makes debugging much easier)
- For enums: #[derive(Debug, Clone, Copy, PartialEq)]  
(enums are usually small, copyable)
- For Tauri: #[derive(Debug, Clone, serde::Serialize)]  
(to send to frontend)
- For state: #[derive(Debug, Default)]  
(makes initialization easier)

---

## 8. The Module System: Organizing Code

**Keywords:**

- **mod** = Declare a module
- **pub** = Make something public
- **use** = Bring something into scope

**From our codebase:**

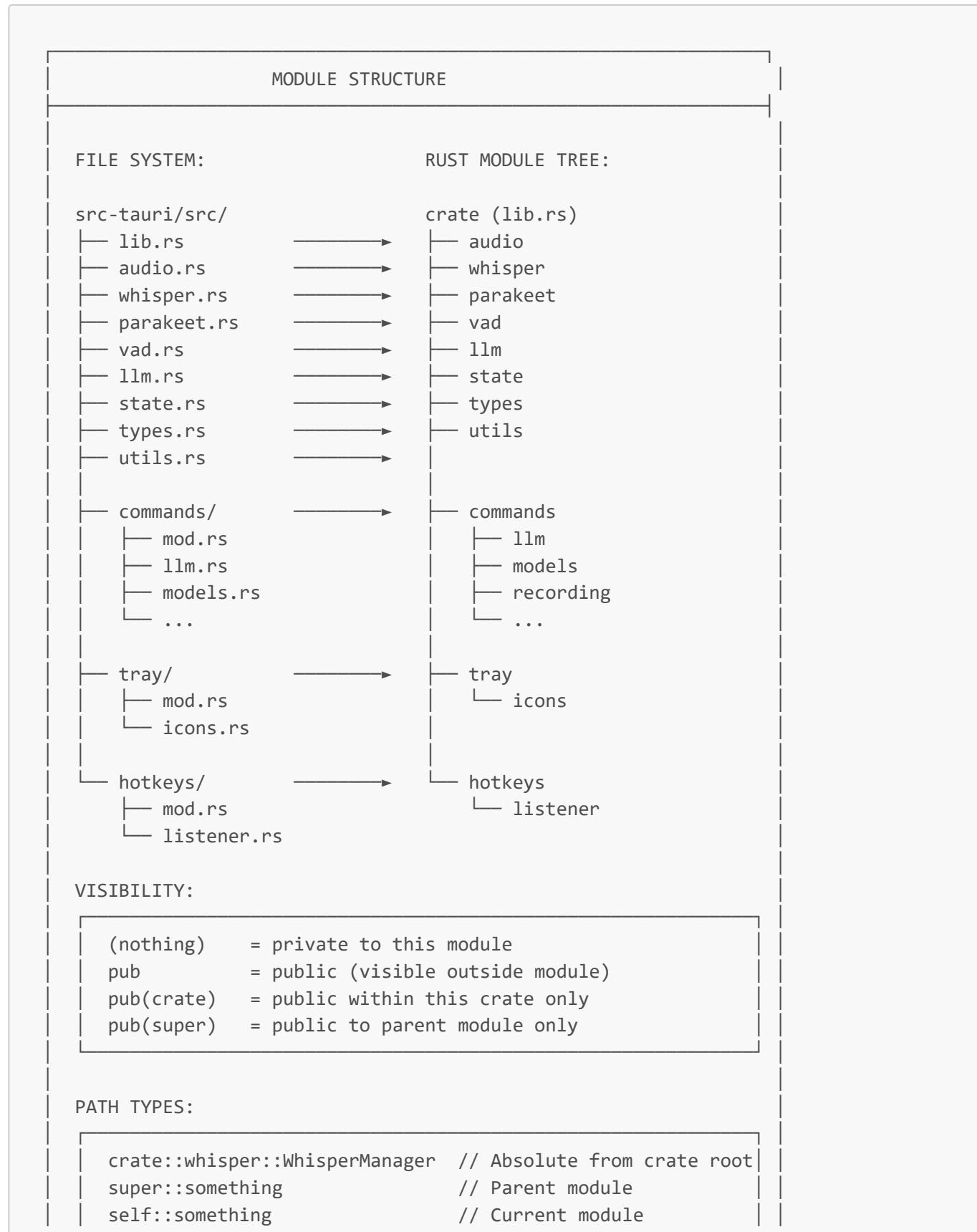
```
// In lib.rs - Declaring modules
mod audio;           // Loads from audio.rs
mod commands;        // Loads from commands/mod.rs (it's a folder!)
mod whisper;
mod parakeet;

// In commands/mod.rs - Re-exporting
pub mod l1m;
pub mod models;
pub mod recording;
```

```
// Now commands::recording::start_recording is accessible

// In lib.rs - Using items
use crate::commands::*;

use crate::whisper::WhisperManager; // Import specific item
```

**Visual:**

```
    something           // Relative (in scope)
```

## 9. Lifetimes: The Borrow Checker's Memory

**Lifetimes tell the compiler how long references are valid.**

You'll see them as `'a`, `'static`, etc.

From our codebase:

```
// In commands - Lifetime in function signature
#[tauri::command]
pub async fn correct_text(
    state: State<'_, AudioState>, // '_' = "infer the lifetime"
    text: String,
) -> Result<String, String>

// What the lifetime means:
// The State reference is valid for the duration of this function call
// Tauri manages this automatically
```

When you need explicit lifetimes:

### LIFETIMES EXPLAINED

MOST OF THE TIME: You don't need to write lifetimes!  
The compiler infers them (lifetime elision).

```
// Compiler infers lifetimes here:
fn first_word(s: &str) -> &str { ... }

// What compiler actually sees:
fn first_word<'a>(s: &'a str) -> &'a str { ... }
//          ^^^      ^^^        ^
//          |          |          |
//          |          |          |
//          |          |          | Output lives as long
//          |          |          | as input
//          |          |          |
//          |          |          | Input has lifetime 'a
//          |          |          |
//          |          |          | Declare lifetime parameter
```

WHEN YOU NEED EXPLICIT LIFETIMES:

```
// Multiple references - which one does output come from?
fn longer<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() { s1 } else { s2 }
}

// Storing references in structs
struct Excerpt<'a> {
    text: &'a str, // This reference must outlive struct
}
```

'static LIFETIME:

```
// 'static = lives for entire program duration

let s: &'static str = "Hello"; // String literals
// String literals are baked into the binary,
// so they live forever!
```

VISUAL:

```
fn example() {
    let s1 = String::from("hello");
    {
        let s2 = String::from("hi");
        let r = longer(&s1, &s2);
        //           ^^^^^^^^^^^^^^
        // r's lifetime = shorter of
        // s1 and s2 = 'b

        println!("{}", r); // OK
    } // s2 dropped here
    // Can't use r here - s2 is gone!
} // s1 dropped here
```

## 10. **impl** Blocks: Adding Methods

**Two types of functions in **impl**:**

- **Methods:** Take `self` as first parameter (called on instance)
- **Associated functions:** No `self` (called on type, like constructors)

### From our codebase:

```
impl WhisperManager {
    // Associated function (no self) - called as WhisperManager::new()
    pub fn new() -> Self {
        Self {
            context: None,
            // ...
        }
    }

    // Method (takes &self) - called as manager.get_backend()
    pub fn get_backend(&self) -> &GpuBackend {
        &self.backend
    }

    // Method (takes &mut self) - called as manager.initialize()
    pub fn initialize(&mut self, model_id: Option<&str>) -> Result<String, String>
{
    // Can modify self
    self.context = Some(ctx);
    // ...
}

// Associated function (no self) - called as
WhisperManager::list_available_models()
pub fn list_available_models() -> Result<Vec<ModelInfo>, String> {
    // Note: No self parameter!
    // ...
}
}
```

### Visual:

#### METHODS vs ASSOCIATED FUNCTIONS

```
impl MyType {
    // ASSOCIATED FUNCTION (no self)
    // Called: MyType::new()
    pub fn new() -> Self { ... }
    //           ^
    //           No self parameter!

    // METHOD - Immutable borrow
    // Called: instance.read_data()
    pub fn read_data(&self) -> Data { ... }
}
```

```

//          ^^^^^^
//          Borrows self, can't modify

// METHOD - Mutable borrow
// Called: instance.modify_data()
pub fn modify_data(&mut self) { ... }
//          ^^^^^^^^^^
//          Borrows self mutably, can modify

// METHOD - Takes ownership
// Called: instance.consume()
pub fn consume(self) { ... }
//          ^
//          Takes ownership, instance unusable after
}

```

CALLING THEM:

```

// Associated function - use :: syntax
let manager = WhisperManager::new();
//          ^^

// Methods - use . syntax
let backend = manager.get_backend();
//          ^

// Mutable method needs mut variable
let mut manager = WhisperManager::new();
manager.initialize(None)?; // Needs &mut self

```

## 11. Conditional Compilation: #[cfg(...)]

**Compile different code for different platforms/features.**

**From our codebase:**

```

// In whisper.rs - Different code for macOS vs others
#[cfg(target_os = "macos")]
unsafe extern "C" fn null_log_callback(_level: u32, ...) { }
//          ^
//          macOS uses u32

#[cfg(not(target_os = "macos"))]
unsafe extern "C" fn null_log_callback(_level: i32, ...) { }
//          ^
//          Other OSes use i32

```

```
// In parakeet.rs - Feature-gated code
#[cfg(any(
    target_os = "linux",
    all(target_os = "windows", target_arch = "x86_64")
))]
{
    // CUDA code - only on Linux or Windows x64
}

#[cfg(not(any(...)))]
{
    // Fallback code for other platforms
}
```

## Common `cfg` conditions:

### CONDITIONAL COMPIILATION

#### PLATFORM DETECTION:

```
#[cfg(target_os = "windows")]
#[cfg(target_os = "macos")]
#[cfg(target_os = "linux")]

#[cfg(target_arch = "x86_64")]    // Intel/AMD 64-bit
#[cfg(target_arch = "aarch64")]   // ARM 64-bit (M1/M2)

#[cfg(windows)]    // Shorthand for target_os = "windows"
#[cfg(unix)]       // Linux, macOS, BSD, etc.
```

#### COMBINING CONDITIONS:

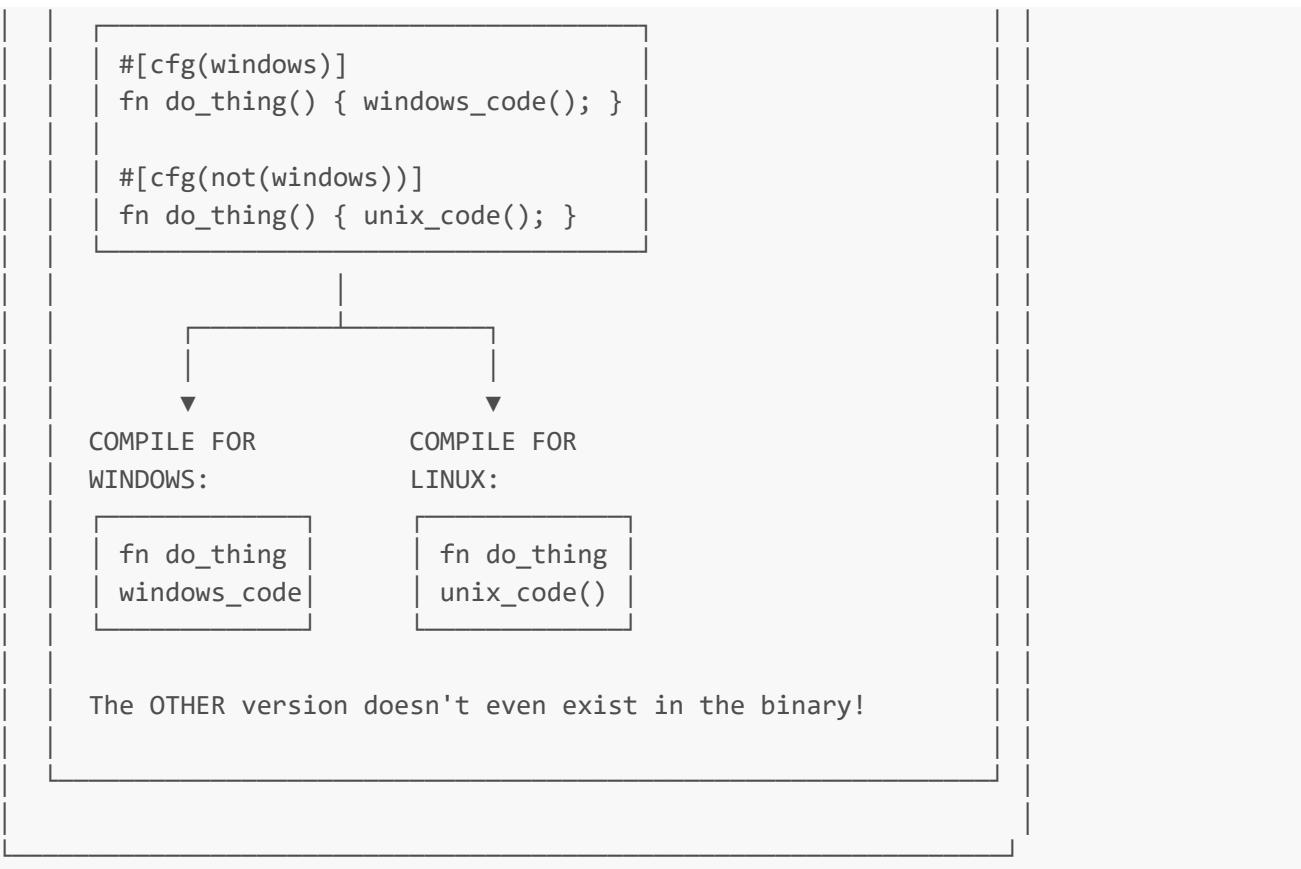
```
#[cfg(not(windows))]           // NOT Windows
#[cfg(any(linux, macos))]      // Linux OR macOS
#[cfg(all(windows, x86_64))]   // Windows AND x86_64
```

#### FEATURE FLAGS:

```
#[cfg(feature = "cuda")]        // If "cuda" feature on
#[cfg(debug_assertions)]         // Debug mode
#[cfg(test)]                     // Running tests
```

#### VISUAL:

SOURCE CODE:



## 12. Common Rust Idioms in This Codebase

### Patterns you'll see repeatedly:

```

// 1. UNWRAP_OR - Default value if None/Err
let model = model_id.unwrap_or("tiny.en-q5_1");
// If model_id is None, use "tiny.en-q5_1"

// 2. OK_OR - Convert Option to Result
let device = host.default_input_device()
    .ok_or("No input device")?;
// If None, return Err("No input device")

// 3. MAP_ERR - Transform error type
let reader = hound::WavReader::open(file_path)
    .map_err(|e| format!("Failed to open: {}", e))?;
// Transform hound::Error into String

// 4. AS_REF - Convert Option<T> to Option<&T>
pub fn get_current_model(&self) -> Option<&String> {
    self.current_model.as_ref()
}
// Don't move out of self, just borrow

// 5. CLONED - Convert Option<&T> to Option<T>
let model_name = whisper.get_current_model().cloned();
// Get owned String from &String

```

```

// 6. INTO - Convert types
let config: StreamConfig = device.default_input_config()?.into();
// Convert SupportedStreamConfig into StreamConfig

// 7. TO_STRING / TO_OWNED - Create owned String from &str
let id = file_name.trim_start_matches("ggml-").to_string();

// 8. FORMAT! - String interpolation
let message = format!("Processing {} files", count);

// 9. COLLECT - Turn iterator into collection
let models: Vec<ModelInfo> = entries.collect();

// 10. CLONE - Explicit deep copy
let whisper = state.whisper.clone(); // Clone the Arc (cheap)

```

### Visual cheat sheet:

RUST IDIOM CHEAT SHEET																									
OPTION METHODS:																									
<table border="1"> <tbody> <tr><td>.unwrap()</td><td>→ T or PANIC!</td></tr> <tr><td>.expect("msg")</td><td>→ T or PANIC with message</td></tr> <tr><td>.unwrap_or(default)</td><td>→ T or default value</td></tr> <tr><td>.unwrap_or_else(f)</td><td>→ T or compute default with closure</td></tr> <tr><td>.ok_or(err)</td><td>→ Result&lt;T, E&gt;</td></tr> <tr><td>.map(f)</td><td>→ Option&lt;U&gt; (transform inner value)</td></tr> <tr><td>.and_then(f)</td><td>→ Option&lt;U&gt; (chain Option operations)</td></tr> <tr><td>.as_ref()</td><td>→ Option&lt;&amp;T&gt;</td></tr> <tr><td>.as_mut()</td><td>→ Option&lt;&amp;mut T&gt;</td></tr> <tr><td>.cloned()</td><td>→ Option&lt;T&gt; (clone inner value)</td></tr> <tr><td>.is_some()</td><td>→ bool</td></tr> <tr><td>.is_none()</td><td>→ bool</td></tr> </tbody> </table>		.unwrap()	→ T or PANIC!	.expect("msg")	→ T or PANIC with message	.unwrap_or(default)	→ T or default value	.unwrap_or_else(f)	→ T or compute default with closure	.ok_or(err)	→ Result<T, E>	.map(f)	→ Option<U> (transform inner value)	.and_then(f)	→ Option<U> (chain Option operations)	.as_ref()	→ Option<&T>	.as_mut()	→ Option<&mut T>	.cloned()	→ Option<T> (clone inner value)	.is_some()	→ bool	.is_none()	→ bool
.unwrap()	→ T or PANIC!																								
.expect("msg")	→ T or PANIC with message																								
.unwrap_or(default)	→ T or default value																								
.unwrap_or_else(f)	→ T or compute default with closure																								
.ok_or(err)	→ Result<T, E>																								
.map(f)	→ Option<U> (transform inner value)																								
.and_then(f)	→ Option<U> (chain Option operations)																								
.as_ref()	→ Option<&T>																								
.as_mut()	→ Option<&mut T>																								
.cloned()	→ Option<T> (clone inner value)																								
.is_some()	→ bool																								
.is_none()	→ bool																								
RESULT METHODS:																									
<table border="1"> <tbody> <tr><td>.unwrap()</td><td>→ T or PANIC with error</td></tr> <tr><td>.expect("msg")</td><td>→ T or PANIC with message</td></tr> <tr><td>.unwrap_or(default)</td><td>→ T or default value</td></tr> <tr><td>.ok()</td><td>→ Option&lt;T&gt; (discard error)</td></tr> <tr><td>.err()</td><td>→ Option&lt;E&gt; (discard success)</td></tr> <tr><td>.map(f)</td><td>→ Result&lt;U, E&gt;</td></tr> <tr><td>.map_err(f)</td><td>→ Result&lt;T, F&gt; (transform error)</td></tr> <tr><td>.and_then(f)</td><td>→ Result&lt;U, E&gt;</td></tr> <tr><td>.is_ok()</td><td>→ bool</td></tr> <tr><td>.is_err()</td><td>→ bool</td></tr> <tr><td>?</td><td>→ Early return on Err</td></tr> </tbody> </table>		.unwrap()	→ T or PANIC with error	.expect("msg")	→ T or PANIC with message	.unwrap_or(default)	→ T or default value	.ok()	→ Option<T> (discard error)	.err()	→ Option<E> (discard success)	.map(f)	→ Result<U, E>	.map_err(f)	→ Result<T, F> (transform error)	.and_then(f)	→ Result<U, E>	.is_ok()	→ bool	.is_err()	→ bool	?	→ Early return on Err		
.unwrap()	→ T or PANIC with error																								
.expect("msg")	→ T or PANIC with message																								
.unwrap_or(default)	→ T or default value																								
.ok()	→ Option<T> (discard error)																								
.err()	→ Option<E> (discard success)																								
.map(f)	→ Result<U, E>																								
.map_err(f)	→ Result<T, F> (transform error)																								
.and_then(f)	→ Result<U, E>																								
.is_ok()	→ bool																								
.is_err()	→ bool																								
?	→ Early return on Err																								

## STRING CONVERSIONS:

```
&str → String:  
    "hello".to_string()  
    "hello".to_owned()  
    String::from("hello")  
    "hello".into()  
  
String → &str:  
    &my_string  
    my_string.as_str()  
  
PathBuf → &Path:  
    path_buf.as_path()  
    &path_buf  
  
&Path → PathBuf:  
    path.to_path_buf()  
    PathBuf::from(path)
```

## 13. Common Beginner Gotchas

### Mistakes you'll make (and how to fix them):

#### COMMON MISTAKES

##### GOTCHA #1: "Cannot move out of borrowed content"

```
// ✗ WRONG  
fn get_name(&self) -> String {  
    self.name // Error! Can't move out of &self  
}  
  
// ☑ FIX: Clone or return reference  
fn get_name(&self) -> String {  
    self.name.clone() // Clone it  
}  
// OR  
fn get_name(&self) -> &String {  
    &self.name // Return reference  
}
```

##### GOTCHA #2: "Value borrowed here after move"

```
// ❌ WRONG
let data = vec![1, 2, 3];
process(data);           // data moved here
println!("{:?}", data); // Error! data was moved

// ✅ FIX: Borrow instead of move
let data = vec![1, 2, 3];
process(&data);         // Borrow, don't move
println!("{:?}", data); // Works!

// OR clone if you need to give ownership
process(data.clone());
println!("{:?}", data); // Works!
```

## GOTCHA #3: "Cannot borrow as mutable more than once"

```
// ❌ WRONG
let mut data = vec![1, 2, 3];
let r1 = &mut data;
let r2 = &mut data; // Error! Already borrowed mutably
r1.push(4);
r2.push(5);

// ✅ FIX: Scope the borrows
let mut data = vec![1, 2, 3];
{
    let r1 = &mut data;
    r1.push(4);
} // r1 goes out of scope
{
    let r2 = &mut data;
    r2.push(5);
}
```

## GOTCHA #4: "Mutex poisoned"

```
// Happens when a thread panics while holding the lock

// ❌ Can crash
let data = mutex.lock().unwrap();

// ✅ Handle poison error
let data = mutex.lock().unwrap_or_else(|e| e.into_inner());
// OR just use unwrap if panic = unrecoverable anyway
```

## GOTCHA #5: "Expected (), found String"

```
// ❌ WRONG - Last expression has ;
fn get_message() -> String {
    "hello".to_string(); // Semicolon makes it statement
```

```

        } // Returns () not String!

// ✅ FIX - Remove semicolon for implicit return
fn get_message() -> String {
    "hello".to_string() // No semicolon = return this
}

```

#### GOTCHA #6: "Use of moved value in closure"

```

// ❌ WRONG
let data = vec![1, 2, 3];
std::thread::spawn(|| {
    println!("{}:?", data); // Error! May outlive data
});

// ✅ FIX - Use move closure
let data = vec![1, 2, 3];
std::thread::spawn(move || {
    println!("{}:?", data); // data moved into closure
});

```

## 14. Memory Layout: Stack vs Heap

### Understanding where data lives:

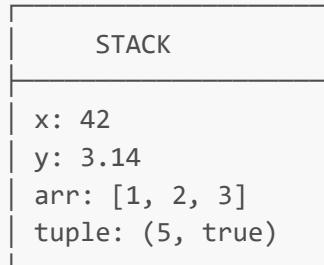
#### STACK vs HEAP

#### THE STACK (Fast, Fixed-Size, Automatic):

```

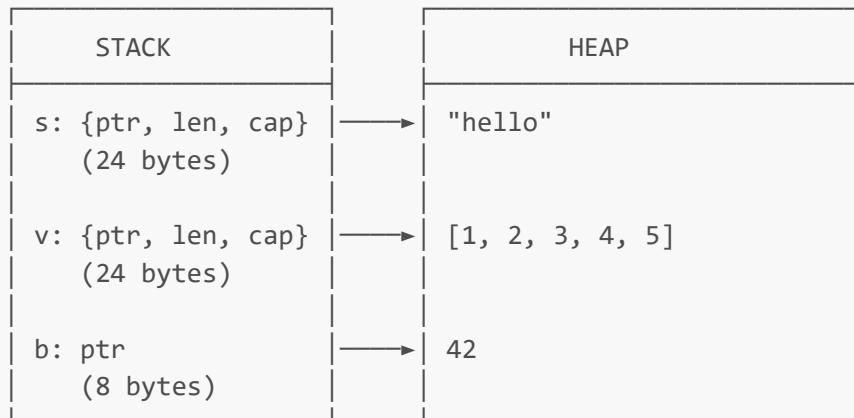
let x: i32 = 42;           // 4 bytes on stack
let y: f64 = 3.14;         // 8 bytes on stack
let arr: [i32; 3] = [1,2,3]; // 12 bytes on stack
let tuple: (i32, bool) = (5, true); // 5 bytes on stack

```



### THE HEAP (Flexible, Dynamic, Manual-ish):

```
let s: String = String::from("hello");
let v: Vec<i32> = vec![1, 2, 3, 4, 5];
let b: Box<i32> = Box::new(42);
```



↑ Stack has POINTERS  
to heap data

↑ Heap has ACTUAL DATA  
(can grow/shrink)

### WHEN DATA IS DROPPED:

```
fn example() {
    let s = String::from("hello");
    // s is on stack, "hello" is on heap
} // s goes out of scope:
// 1. Rust calls drop(s)
// 2. Heap memory for "hello" is freed
// 3. Stack memory for s is reclaimed

// No memory leaks! No manual free()!
```

### WHY THIS MATTERS FOR OUR CODE:

```
// Audio data is BIG - must be on heap
let audio: Vec<f32> = vec![0.0; 480000]; // ~2MB!

// WhisperContext is BIG - Box it for cheaper moves
let resampler: Box<SincFixedIn<f32>> = Box::new(...);

// Arc means multiple owners of SAME heap data
let whisper: Arc<Mutex<WhisperManager>> = Arc::new(...);
let whisper2 = whisper.clone(); // Same heap data!
```

## 15. Reading Rust Error Messages

**Rust errors are verbose but helpful. Learn to read them!**

### READING ERROR MESSAGES

#### EXAMPLE ERROR:

```
error[E0382]: borrow of moved value: `data`
--> src/main.rs:5:20
|
3 |     let data = vec![1, 2, 3];
|         ----- move occurs because `data` has type
|             `Vec<i32>`, which does not implement
|                 the `Copy` trait
4 |     process(data);
|             ----- value moved here
5 |     println!("{}:", data);
|             ^^^^^ value borrowed here
|                 after move
```

For more information about this error, try `rustc --explain E0382`.

#### HOW TO READ IT:

1. ERROR CODE: E0382  
Run `rustc --explain E0382` for detailed explanation
2. WHAT HAPPENED: "borrow of moved value"  
You tried to use something after giving it away
3. WHERE: src/main.rs:5:20  
File, line 5, column 20
4. THE STORY (read the ---- arrows):  
Line 3: data created  
Line 4: data MOVED into process()  
Line 5: tried to use data (ERROR!)
5. WHY: Vec<i32> doesn't implement Copy  
So passing it moves ownership

## COMMON ERROR CODES:

```
E0382 - Use of moved value
E0502 - Cannot borrow as mutable (already borrowed)
E0499 - Cannot borrow mutably more than once
E0597 - Borrowed value does not live long enough
E0106 - Missing lifetime specifier
E0308 - Mismatched types
E0425 - Cannot find value in scope
E0433 - Failed to resolve (module/type not found)
```

 Quick Reference Card

## RUST QUICK REFERENCE

## OWNERSHIP:

```
let x = value;          // x owns value
let y = x;              // ownership moves to y, x invalid
let z = x.clone();     // z gets a copy, x still valid
```

## BORROWING:

```
&value           // immutable borrow (can read)
&mut value      // mutable borrow (can modify)
```

## ERROR HANDLING:

```
Option<T>        // Some(value) or None
Result<T, E>      // Ok(value) or Err(error)
value?            // return early if Err/None
value.unwrap()    // get value or panic
```

## PATTERN MATCHING:

```
match value { ... } // exhaustive matching
if let Some(x) = opt // match single pattern
while let Ok(x) = iter // loop while pattern matches
```

## CLOSURES:

```
|x| x + 1          // simple closure
|x, y| { ... }    // multi-line closure
move |x| ...       // take ownership of captures
```

## SMART POINTERS:

```
Box<T>            // heap allocation
Rc<T>             // reference counting (single-thread)
Arc<T>             // atomic ref counting (multi-thread)
```

```

Mutex<T>           // mutual exclusion lock

COMMON TRAITS:
Debug                // {:?} formatting
Clone                // .clone() method
Copy                 // implicit copying
Send                 // can send between threads
Sync                 // can share between threads

MODULES:
mod name;           // declare module
pub                 // make public
use crate::path::Item; // bring into scope

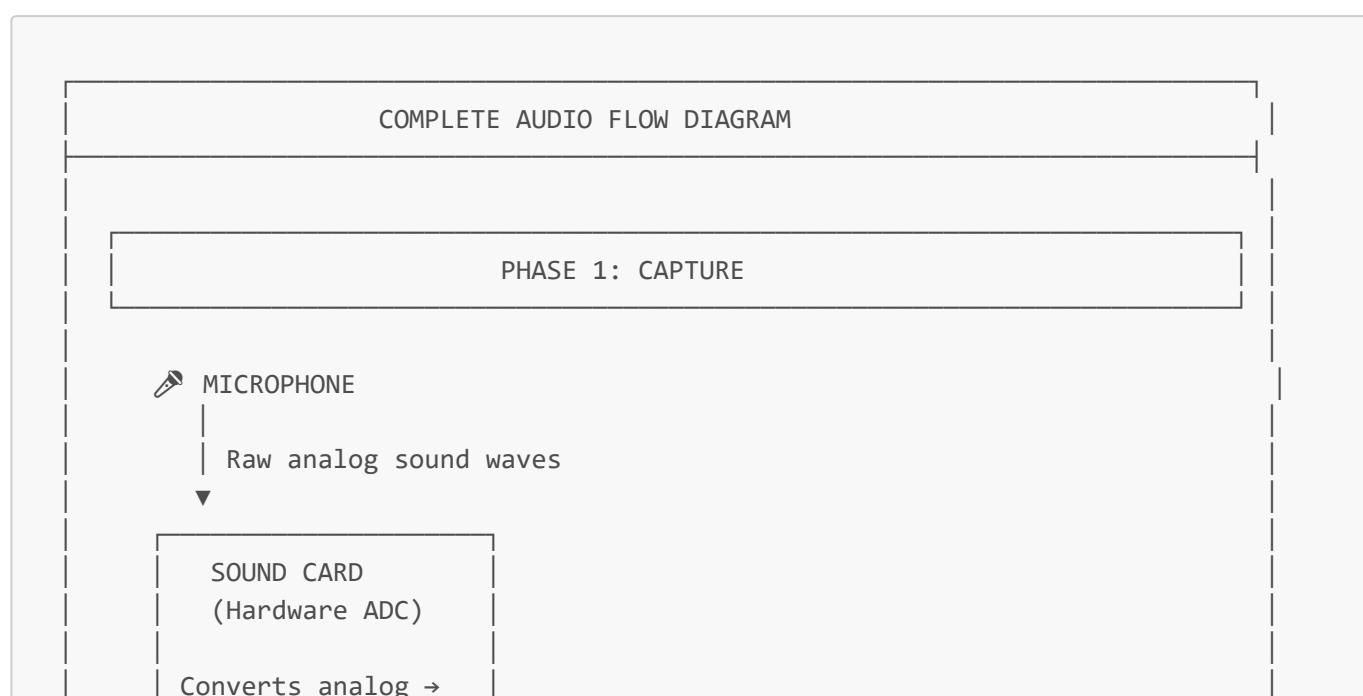
USEFUL METHODS:
.to_string()          // &str → String
.as_str()             // String → &str
.clone()              // deep copy
.into()               // convert types
.as_ref()              // Option<T> → Option<&T>
.map()                // transform inner value
.map_err()             // transform error
.ok_or()               // Option → Result
.collect()              // iterator → collection

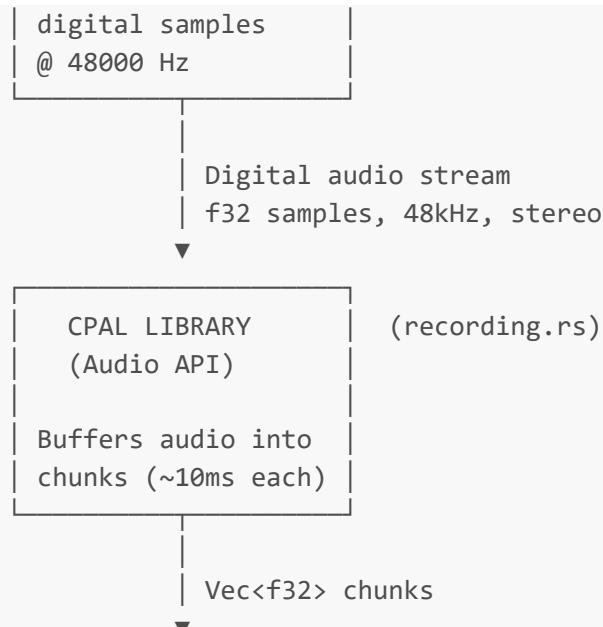
```

## 🔊 Audio Flow Diagrams

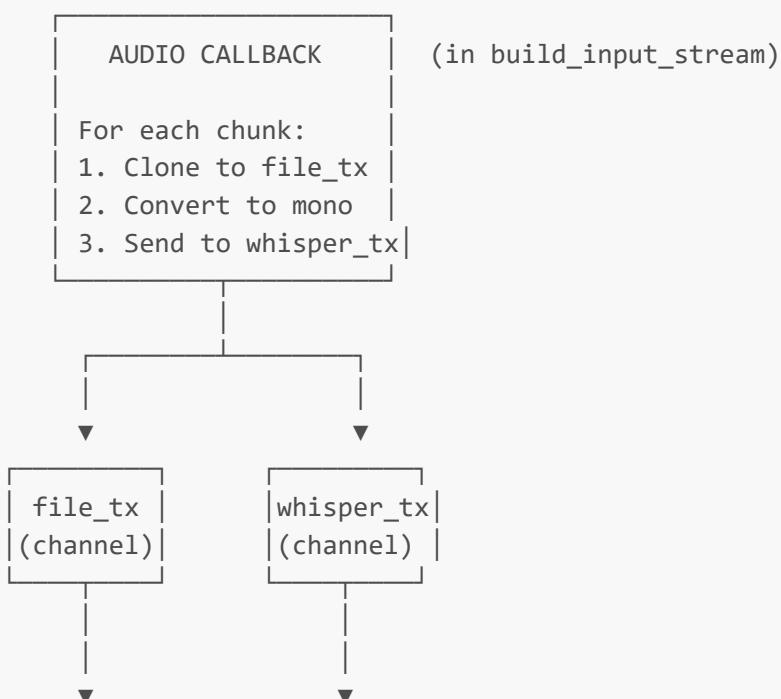
This section shows how audio data flows through the entire system, from microphone to final transcript.

### Complete Recording Flow

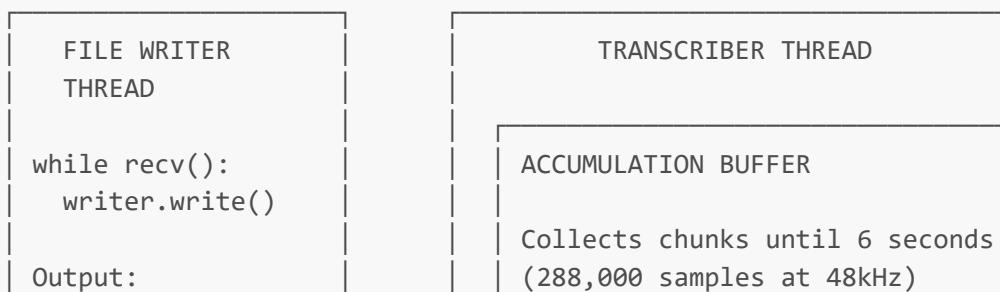


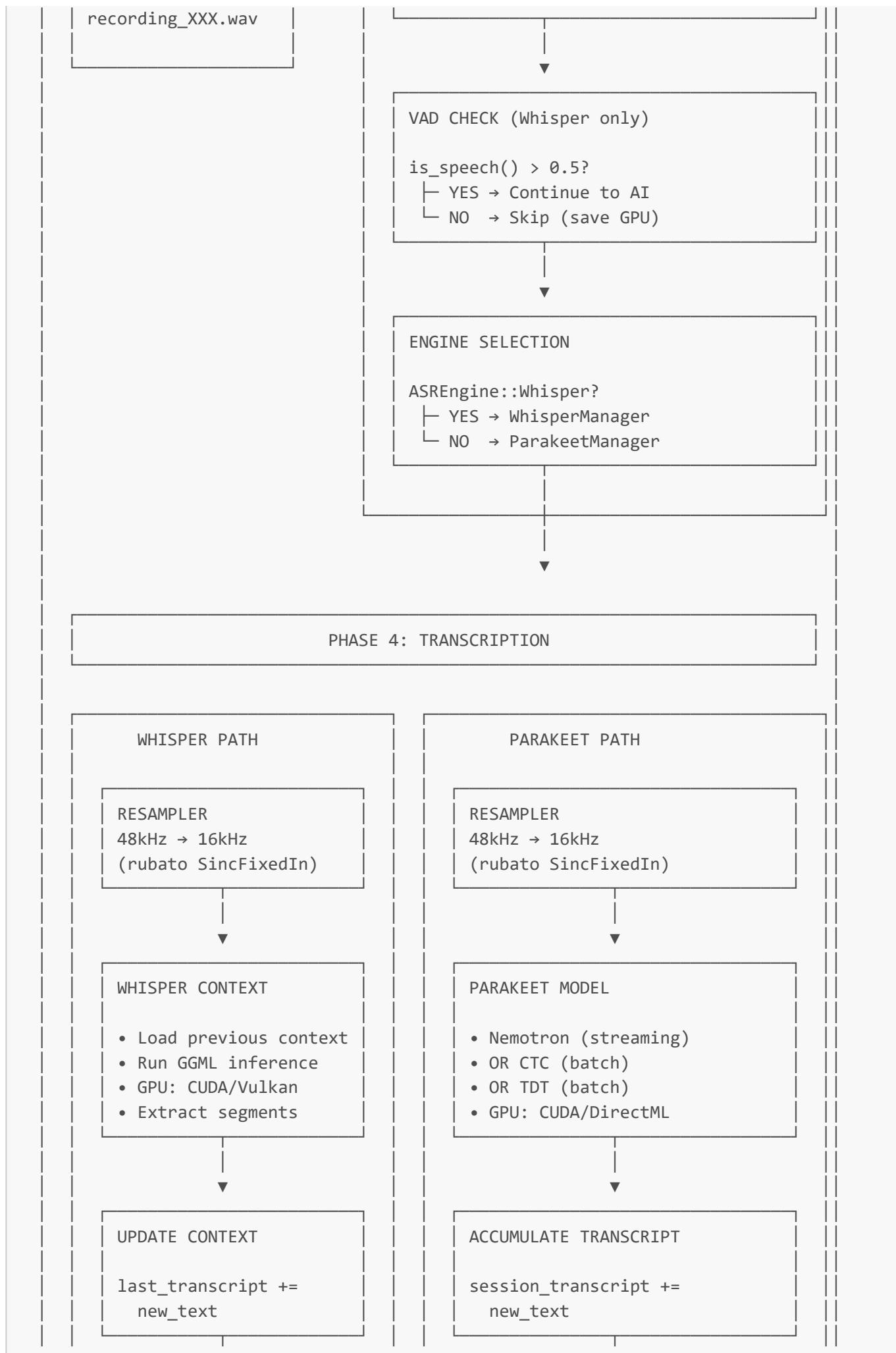


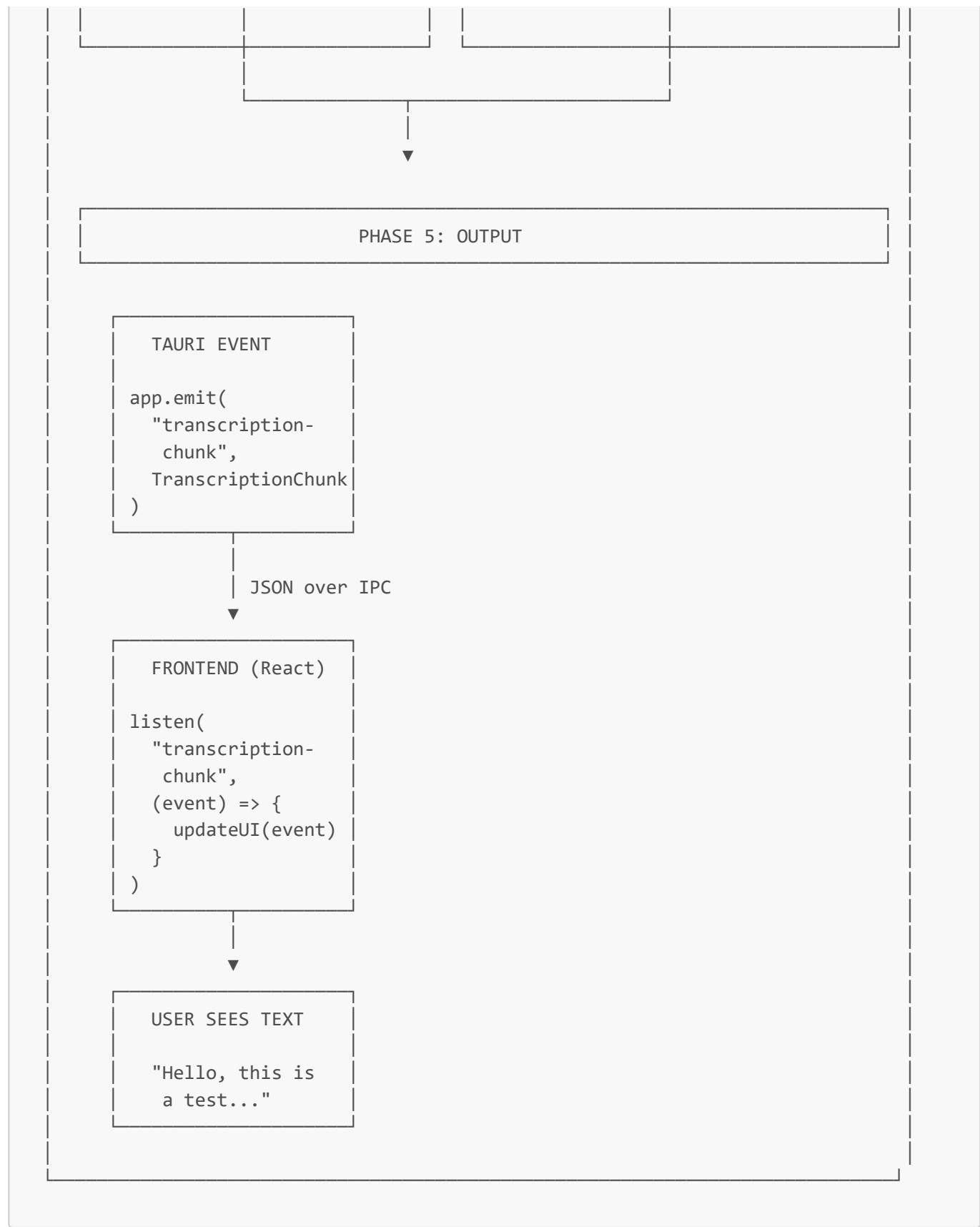
## PHASE 2: DISTRIBUTION



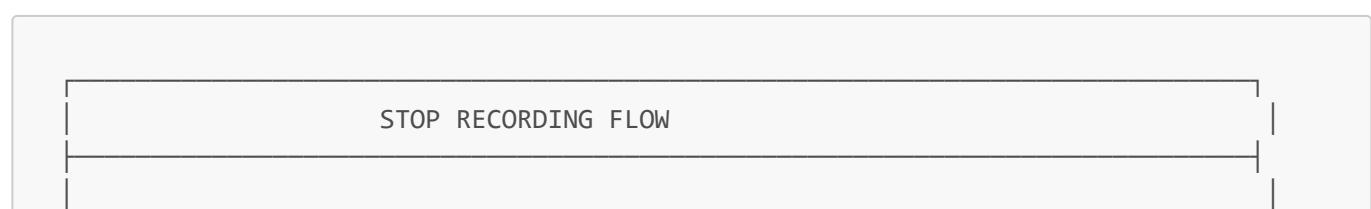
## PHASE 3: PROCESSING

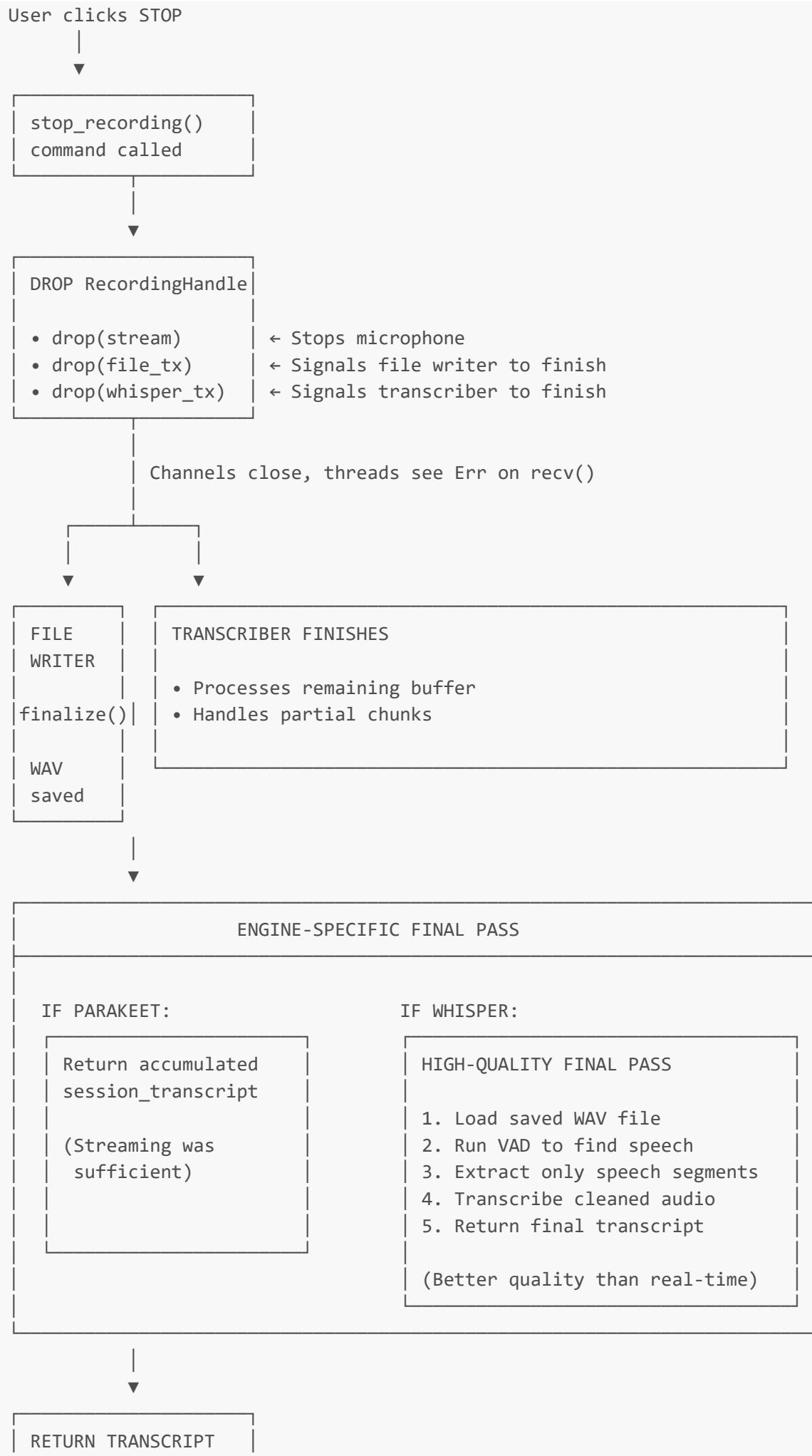






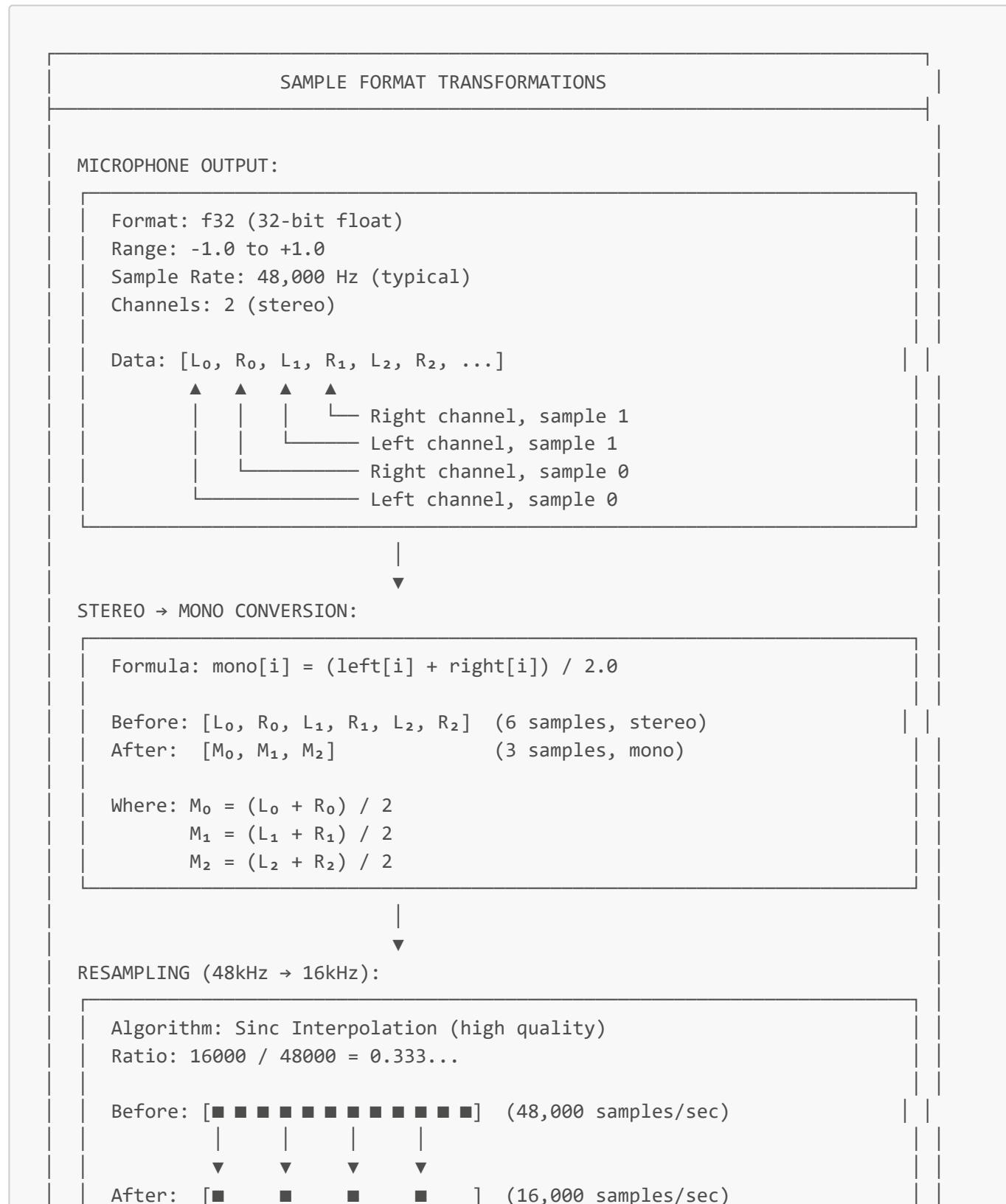
## Stop Recording Flow





```
Ok("Final cleaned  
transcript...")
```

## Audio Sample Format Flow



Note: Not just dropping samples! Sinc filter preserves audio quality.  
It's like resizing an image with anti-aliasing.



#### WHISPER/PARAKEET INPUT:

Format: f32 (32-bit float)  
Range: -1.0 to +1.0  
Sample Rate: 16,000 Hz (required!)  
Channels: 1 (mono)

This is what the AI models were trained on.

#### NUMBERS:

1 second of audio:

Stage	Samples	Bytes (f32)	Bytes (i16)
Mic (48kHz stereo)	96,000	384,000	192,000
Mono (48kHz)	48,000	192,000	96,000
Resampled (16kHz)	16,000	64,000	32,000

6 second chunk (what we process):

Mic (48kHz stereo) 576,000 2,304,000 bytes (~2.2 MB)  
Final (16kHz mono) 96,000 384,000 bytes (~375 KB)

## VAD (Voice Activity Detection) Flow

### VAD DECISION FLOW

INPUT: 6 seconds of audio (96,000 samples at 16kHz)

STEP 1: Calculate Energy (RMS)

$$\text{RMS} = \sqrt{(\sum(\text{sample}^2)) / n}$$

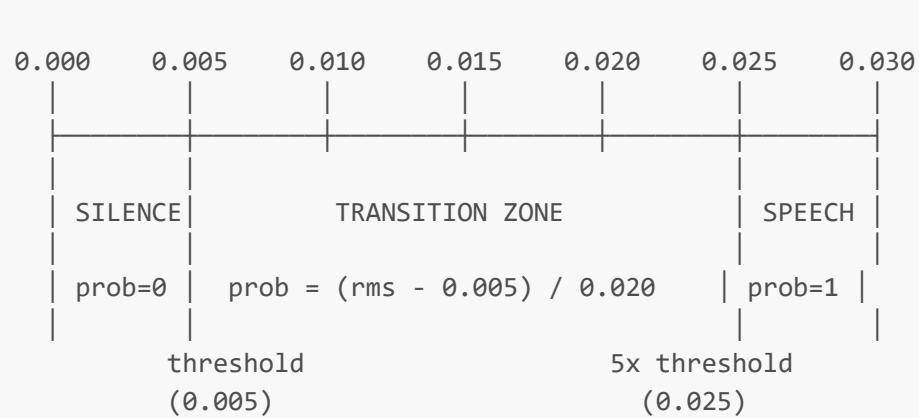
Example:

```
samples = [0.1, -0.2, 0.15, -0.1]
```

```
squares = [0.01, 0.04, 0.0225, 0.01]
mean = 0.0825 / 4 = 0.020625
RMS = sqrt(0.020625) = 0.1436
```

#### STEP 2: Compare to Threshold

threshold = 0.005



### STEP 3: Make Decision

```
if probability > 0.5:  
    → PROCESS with Whisper AI (expensive)  
else:  
    → SKIP this chunk (save GPU cycles)
```

## TYPICAL RECORDING:

The figure shows a timeline from 0s to 54s. The 'Audio' row displays spectrograms at 6-second intervals. The 'VAD' row shows labels below each spectrogram: SKIP, PROC, PROC, SKIP, SKIP, PROC, SKIP, PROC, PROC, SKIP.

Time	0s	6s	12s	18s	24s	30s	36s	42s	48s	54s
Audio										
VAD	SKIP	PROC	PROC	SKIP	SKIP	PROC	SKIP	PROC	PROC	SKIP

Result: Only 5 out of 10 chunks processed!  
50% GPU savings!

 = Silence (background noise, pauses)  
 = Speech (actual talking)

## 🔗 Tauri Command API Reference

Every command exposed to the frontend, with types and examples.

### Recording Commands

#### start_recording

Property	Value
File	commands/recording.rs
Purpose	Initialize microphone and begin recording
Frontend Call	<code>await invoke("start_recording")</code>

#### Signature:

```
#[tauri::command]
pub fn start_recording(
    app_handle: AppHandle, // Injected by Tauri
    state: State<AudioState>, // Injected by Tauri
) -> Result<String, String>
```

#### Returns:

- `Ok("Recording started: /path/to/recording_1234567890.wav")`
- `Err("No input device")` - No microphone found
- `Err("Failed to create stream: ...")` - Audio system error

#### Frontend Example:

```
import { invoke } from "@tauri-apps/api/core";

async function startRecording() {
    try {
        const result = await invoke<string>("start_recording");
        console.log(result); // "Recording started: ..."
    } catch (error) {
        console.error("Failed to start:", error);
    }
}
```

## stop_recording

Property	Value
File	commands/recording.rs
Purpose	Stop recording and get final transcript
Frontend Call	await invoke("stop_recording")

### Signature:

```
#[tauri::command]
pub fn stop_recording(
    state: State<AudioState>,
) -> Result<String, String>
```

### Returns:

- Ok("Final transcript text here...") - Success with transcript
- Ok("[silence]") - Recording contained no speech
- Ok("Recording saved.") - Saved but no transcript (error during processing)
- Err("Not recording") - Wasn't recording

### Frontend Example:

```
async function stopRecording() {
    try {
        const transcript = await invoke<string>("stop_recording");
        console.log("Transcript:", transcript);
    } catch (error) {
        console.error("Not recording:", error);
    }
}
```

## Model Commands

### list_models

Property	Value
File	commands/models.rs
Purpose	Get available Whisper models
Frontend Call	await invoke("list_models")

### Signature:

```
#[tauri::command]
pub fn list_models() -> Result<Vec<ModelInfo>, String>
```

**Returns:**

```
// Success:
[
    {
        id: "tiny.en-q5_1",
        display_name: "Tiny English (Q5_1)",
        file_name: "ggml-tiny.en-q5_1.bin",
        size_mb: 42.3
    },
    {
        id: "base.en-q5_1",
        display_name: "Base English (Q5_1)",
        file_name: "ggml-base.en-q5_1.bin",
        size_mb: 78.5
    }
    // ...
]

// Error:
"Could not find models directory containing ggml models"
```

**Frontend Example:**

```
interface ModelInfo {
    id: string;
    display_name: string;
    file_name: string;
    size_mb: number;
}

async function getModels() {
    const models = await invoke<ModelInfo[]>("list_models");
    models.forEach(m => console.log(`${m.display_name} (${m.size_mb}MB)`));
}
```

**switch_model**

Property	Value
File	commands/models.rs
Purpose	Change the active Whisper model

Property	Value
<b>Frontend Call</b>	<code>await invoke("switch_model", { modelId: "..." })</code>

**Signature:**

```
#[tauri::command]
pub fn switch_model(
    state: State<AudioState>,
    model_id: String,
) -> Result<String, String>
```

**Returns:**

- `Ok("Backend: CUDA")` - Model loaded with GPU
- `Ok("Backend: CPU")` - Model loaded with CPU fallback
- `Err("Cannot switch models while recording")` - Stop recording first
- `Err("Model file not found: ...")` - Model doesn't exist

**Frontend Example:**

```
async function switchToLargeModel() {
    try {
        const backend = await invoke<string>("switch_model", {
            modelId: "large-v3-turbo-q5_1"
        });
        console.log("Loaded with:", backend);
    } catch (error) {
        console.error("Switch failed:", error);
    }
}
```

**list_parakeet_models**

Property	Value
<b>File</b>	<code>commands/models.rs</code>
<b>Purpose</b>	Get available Parakeet/Nemotron models

**Frontend Call** `await invoke("list_parakeet_models")`

**Returns:**

```
[
    {
        id: "nemotron:parakeet-tdt-0.6b-v2",
```

```

        display_name: "Parakeet TDT - parakeet-tdt-0.6b-v2",
        model_type: "TDT",
        size_mb: 245.7
    }
    // ...
]

```

## init_parakeet

Property	Value
<b>File</b>	commands/models.rs
<b>Purpose</b>	Initialize a Parakeet model
<b>Frontend Call</b>	await invoke("init_parakeet", { modelId?: "..." })

### Signature:

```

#[tauri::command]
pub fn init_parakeet(
    state: State<AudioState>,
    model_id: Option<String>, // Optional - defaults to first available
) -> Result<String, String>

```

### Returns:

- Ok("Loaded Parakeet TDT - ... (CUDA)") - Success
- Err("No Parakeet/Nemotron models found.") - No models installed

## get_parakeet_status

Property	Value
<b>File</b>	commands/models.rs
<b>Purpose</b>	Check Parakeet engine status
<b>Frontend Call</b>	await invoke("get_parakeet_status")

### Returns:

```
{
    loaded: true,
    model_id: "tdt:parakeet-tdt-0.6b-v2",
    model_type: "TDT",
    backend: "CUDA"
}
```

---

## Settings Commands

### get_backend_info

Property	Value
File	commands/settings.rs
Purpose	Get current Whisper GPU backend
Frontend Call	await invoke("get_backend_info")

Returns: "CUDA", "Vulkan", or "CPU"

---

### set_active_engine

Property	Value
File	commands/settings.rs
Purpose	Switch between Whisper and Parakeet
Frontend Call	await invoke("set_active_engine", { engine: "whisper" })

**Parameters:**

- engine: "whisper" - Use Whisper AI
- engine: "parakeet" - Use Parakeet ASR

**Returns:**

- Ok("Engine switched to Whisper")
  - Ok("Engine switched to Parakeet")
  - Err("Unknown engine: xyz")
- 

### get_active_engine

Property	Value
File	commands/settings.rs
Purpose	Get currently active ASR engine
Frontend Call	await invoke("get_active_engine")

Returns: "Whisper" or "Parakeet"

---

### set_tray_state

Property	Value
<b>File</b>	commands/settings.rs
<b>Purpose</b>	Update system tray icon
<b>Frontend Call</b>	await invoke("set_tray_state", { newState: "recording" })

**Parameters:**

- newState: "ready" - Green icon
- newState: "recording" - Red icon
- newState: "processing" - Yellow icon

**LLM Commands****init_llm**

Property	Value
<b>File</b>	commands/llm.rs
<b>Purpose</b>	Load Gemma grammar correction model
<b>Frontend Call</b>	await invoke("init_llm")

**Returns:**

- Ok("Gemma LLM initialized successfully")
- Ok("LLM already initialized")
- Err("Failed to load LLM: ...") - Model file not found or GPU error

**correct_text**

Property	Value
<b>File</b>	commands/llm.rs
<b>Purpose</b>	Fix grammar in transcribed text
<b>Frontend Call</b>	await invoke("correct_text", { text: "..." })

**Signature:**

```
#[tauri::command]
pub async fn correct_text(
    state: State<'_, AudioState>,
    text: String,
) -> Result<String, String>
```

**Example:**

```
const corrected = await invoke<string>("correct_text", {
    text: "i went too the store and buyed some milk"
});
// Returns: "I went to the store and bought some milk."
```

**check_llm_status**

Property	Value
<b>File</b>	commands/llm.rs
<b>Purpose</b>	Check if LLM is loaded
<b>Frontend Call</b>	await invoke("check_llm_status")

**Returns:** true or false**Transcription Commands****list_sample_files**

Property	Value
<b>File</b>	commands/transcription.rs
<b>Purpose</b>	Get available test audio files
<b>Frontend Call</b>	await invoke("list_sample_files")

**Returns:**

```
[  
  { name: "test_audio_1.wav", path: "/full/path/to/test_audio_1.wav" },  
  { name: "test_audio_2.wav", path: "/full/path/to/test_audio_2.wav" }  
]
```

**benchmark_test**

Property	Value
<b>File</b>	commands/transcription.rs
<b>Purpose</b>	Compare Whisper vs Parakeet performance
<b>Frontend Call</b>	await invoke("benchmark_test", { filePath: "..." })

**Returns:** Formatted benchmark report string

---

Events (Backend → Frontend)

### transcription-chunk

Property	Value
<b>Emitted By</b>	Recording thread
<b>Purpose</b>	Send real-time transcription to UI

**Payload:**

```
interface TranscriptionChunk {
    text: string;           // The transcribed text
    processing_time_ms: number; // How long it took
    method: string;         // "Whisper" or "Parakeet"
}
```

**Frontend Listener:**

```
import { listen } from "@tauri-apps/api/event";

await listen<TranscriptionChunk>("transcription-chunk", (event) => {
    console.log("New text:", event.payload.text);
    console.log("Took:", event.payload.processing_time_ms, "ms");
});
```

### hotkey-start-recording / hotkey-stop-recording

Property	Value
<b>Emitted By</b>	Hotkey listener thread
<b>Purpose</b>	Notify frontend of global hotkey press

**Frontend Listener:**

```
await listen("hotkey-start-recording", () => {
    console.log("User pressed Ctrl+Win, starting...");
    invoke("start_recording");
});

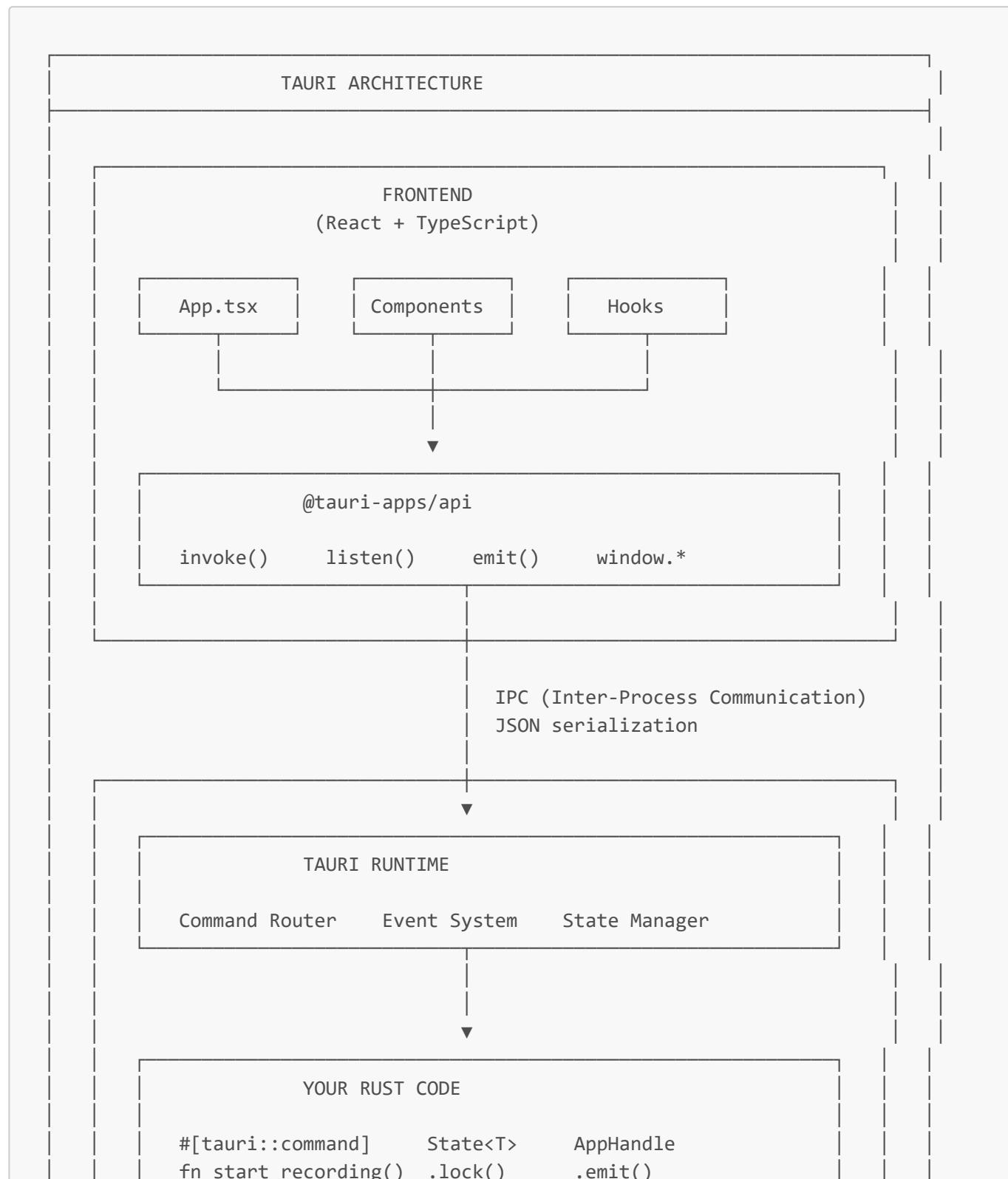
await listen("hotkey-stop-recording", () => {
    console.log("User released Ctrl+Win, stopping...");
```

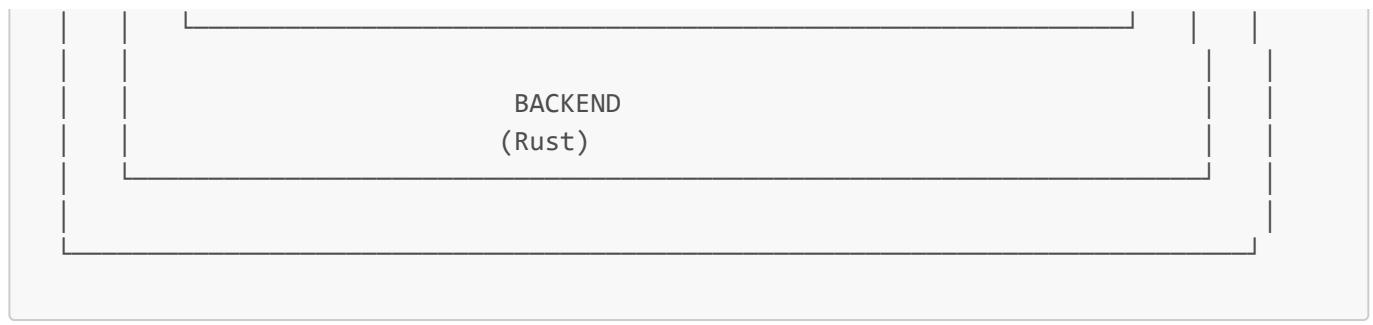
```
    invoke("stop_recording");
});
```

## 🏗 Frontend-Backend Bridge

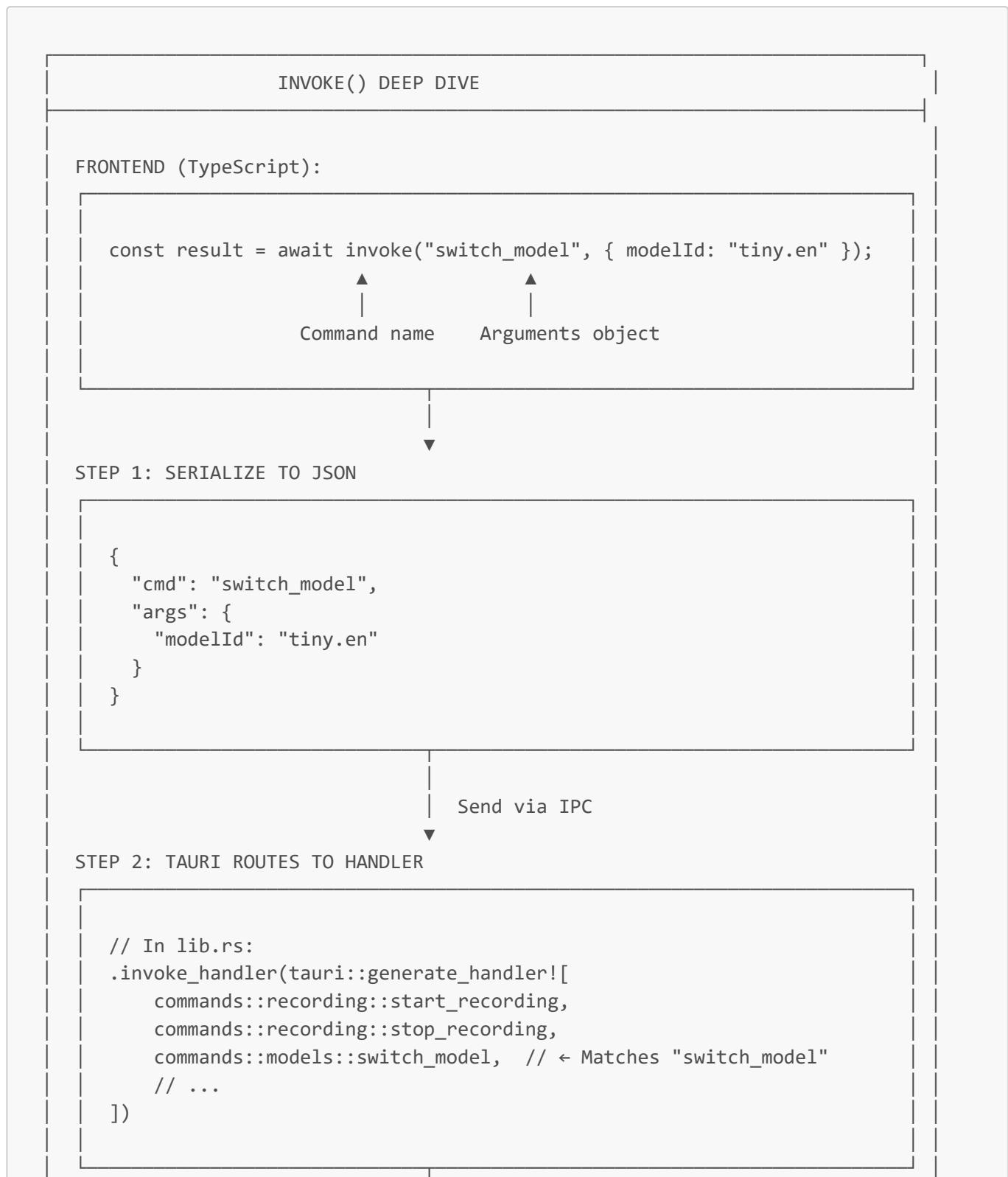
How Tauri connects React (frontend) to Rust (backend).

### The Big Picture





## How `invoke()` Works



↓  
STEP 3: DESERIALIZE ARGUMENTS

```
#[tauri::command]
pub fn switch_model(
    state: State<AudioState>, // Injected by Tauri
    model_id: String,         // ← Deserialized from "modelId"
) -> Result<String, String>
```

Note: camelCase in JS → snake_case in Rust (automatic!)  
"modelId" → model_id

↓  
STEP 4: EXECUTE RUST CODE

```
// Your actual logic runs here
let mut whisper = state.whisper.lock().unwrap();
whisper.initialize(Some(&model_id))?;
Ok(format!("Backend: {}", whisper.get_backend()))
```

↓  
STEP 5: SERIALIZE RESULT

```
Ok("Backend: CUDA") → { "Ok": "Backend: CUDA" }
Err("Not found")     → { "Err": "Not found" }
```

↓ Send via IPC  
↓

STEP 6: PROMISE RESOLVES/REJECTS

```
// Back in TypeScript:
const result = await invoke("switch_model", { modelId: "tiny.en" });
// result = "Backend: CUDA"

// If Err was returned:
// Promise rejects, caught by try/catch
```

## How Events Work (Backend → Frontend)

EVENT SYSTEM

BACKEND EMITS EVENT:

```
// In recording.rs:  
app_handle.emit(  
    "transcription-chunk",      // Event name  
    TranscriptionChunk {        // Payload (must be Serialize)  
        text: "Hello world".to_string(),  
        processing_time_ms: 150,  
        method: "Whisper".to_string(),  
    }  
);
```

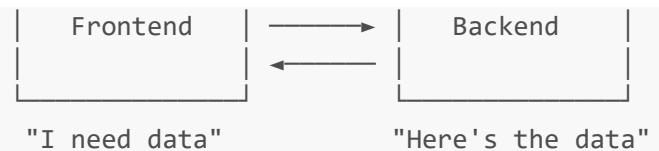
Serialized to JSON  
Sent via IPC

FRONTEND RECEIVES EVENT:

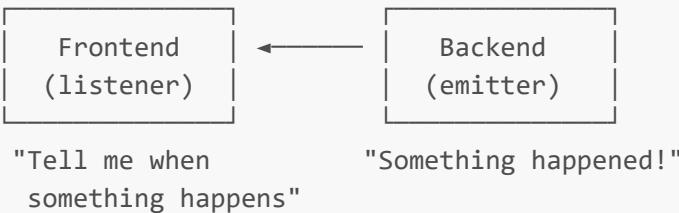
```
import { listen } from "@tauri-apps/api/event";  
  
// Set up listener (usually in useEffect):  
const unlisten = await listen<TranscriptionChunk>(  
    "transcription-chunk",  
    (event) => {  
        console.log("Received:", event.payload);  
        // event.payload = {  
        //     text: "Hello world",  
        //     processing_time_ms: 150,  
        //     method: "Whisper"  
        // }  
        setTranscript(prev => prev + " " + event.payload.text);  
    }  
);  
  
// Clean up when component unmounts:  
return () => { unlisten(); };
```

INVOKE vs EMIT:

INVOKE (Request/Response):



EMIT (Push/Subscribe):



Use INVOKE for: User actions, fetching data, one-time operations

Use EMIT for: Real-time updates, progress, streaming data

## State Management

### TAURI STATE MANAGEMENT

SETUP (in lib.rs):

```

// Create the shared state
let audio_state = AudioState::new();

tauri::Builder::default()
    .manage(audio_state) // ← Register state with Tauri
    .invoke_handler(...)
    .run(...)
  
```

ACCESS IN COMMANDS:

```

#[tauri::command]
pub fn start_recording(
    state: State<AudioState>, // Tauri injects this automatically!
) -> Result<String, String> {

    // Access inner fields (need to lock the Mutex):
    let mut whisper = state.whisper.lock().unwrap();
    whisper.clear_context();
  
```

```
// Or just read:  
let engine = *state.active_engine.lock().unwrap();  
  
Ok("Started".to_string())  
}
```

## WHY THIS PATTERN?

Problem: Multiple commands need access to the same data

✗ Global variables:

```
static mut WHISPER: WhisperManager = ...; // Unsafe!
```

✗ Pass through invoke:

Can't send WhisperManager from frontend!

✓ Tauri State:

- Lives for lifetime of app
- Automatically injected into commands
- Thread-safe via Arc<Mutex<T>>
- Type-safe access

## STATE STRUCTURE:

```
AudioState {  
    whisper: Arc<Mutex<WhisperManager>>  
        └── context: Option<WhisperContext>  
        └── last_transcript: String  
        └── ...  
  
    parakeet: Arc<Mutex<ParakeetManager>>  
        └── model: Option<LoadedModel>  
  
    vad: Arc<Mutex<VADManager>>  
        └── threshold: f32  
  
    recording_handle: Arc<Mutex<Option<RecordingHandle>>>  
        └── None when not recording, Some(...) when recording  
  
    active_engine: Arc<Mutex<ASREngine>>  
        └── Whisper or Parakeet  
}
```

## Serialization Rules

### SERIALIZATION (Rust ↔ JavaScript)

RUST TYPE	JSON	TYPESCRIPT
String	"hello"	string
&str	"hello"	string
i32, u32, i64, u64	123	number
f32, f64	3.14	number
bool	true/false	boolean
()	null	null
Option<T>::Some(v)	v	T   null
Option<T>::None	null	T   null
Vec<T>	[...]	T[]
HashMap<K,V>	{...}	Record<K, V>
Result<T,E>::Ok(v)	v	T (success)
Result<T,E>::Err(e)	throws	catch (e)

### NAMING CONVERSION:

Rust (snake_case)	JavaScript (camelCase)
model_id	modelId
processing_time_ms	processingTimeMs
file_name	fileName

This happens automatically with serde + Tauri!

### STRUCT EXAMPLE:

```
// Rust:
#[derive(serde::Serialize)]
pub struct TranscriptionChunk {
    pub text: String,
    pub processing_time_ms: u32,
    pub method: String,
}

// JSON:
{
    "text": "Hello world",
    "processingTimeMs": 150,
    "method": "Whisper"
}
```

```
// TypeScript:  
interface TranscriptionChunk {  
    text: string;  
    processingTimeMs: number;  
    method: string;  
}
```

ENUM EXAMPLE:

```
// Rust:  
#[derive(serde::Serialize)]  
pub enum ASREngine {  
    Whisper,  
    Parakeet,  
}  
  
// JSON:  
"Whisper" or "Parakeet"  
  
// TypeScript:  
type ASREngine = "Whisper" | "Parakeet";
```

## 📖 Glossary

Technical terms used in this codebase, explained.

### A

**ADC (Analog-to-Digital Converter)** Hardware that converts analog sound waves from a microphone into digital numbers (samples). Your sound card does this.

**Arc (Atomic Reference Counting)** A Rust smart pointer that allows multiple owners of the same data across threads. The "Atomic" means it's thread-safe. See: [Arc<Mutex<WhisperManager>>](#)

**ASR (Automatic Speech Recognition)** The technology of converting spoken audio into text. Whisper and Parakeet are both ASR systems.

### B

**Backend** In this app: The Rust code that handles AI processing. In Whisper context: The hardware running inference (CUDA/Vulkan/CPU).

---

**Borrow** Rust's way of temporarily using data without taking ownership. `&T` is an immutable borrow, `&mut T` is a mutable borrow.

## C

**Channel** A communication pipe between threads. One end sends data (`tx`), the other receives (`rx`). Like a conveyor belt.

**Closure** An anonymous function that can capture variables from its environment. Syntax: `| args | body`

**Context (Whisper)** The "memory" of previous transcriptions. Helps Whisper understand ambiguous words by knowing what came before.

**cpal** Cross-Platform Audio Library. The Rust crate that handles microphone input across Windows/Mac/Linux.

**CTC (Connectionist Temporal Classification)** A type of neural network architecture for sequence-to-sequence tasks like speech recognition. Used by Parakeet CTC models.

**CUDA** NVIDIA's GPU computing platform. Allows running AI models on NVIDIA graphics cards for massive speedup.

---

## D

**Derive** Rust macro that auto-implements traits. `#[derive(Debug, Clone)]` gives your type `debug` formatting and `.clone()`.

**DirectML** Microsoft's machine learning API for Windows. Allows running AI on AMD/Intel/Qualcomm GPUs on Windows.

---

## E

**Enum** A type that can be one of several variants. Like `GpuBackend::Cuda` | `GpuBackend::Vulkan` | `GpuBackend::Cpu`.

---

## F

**f32 / f64** 32-bit and 64-bit floating point numbers. Audio samples are typically f32 (range -1.0 to 1.0).

**Frontend** The user interface part of the app. In Taurscribe: React + TypeScript running in a webview.

---

## G

**GGML / GGUF** File formats for quantized AI models created by Georgi Gerganov. GGUF is the newer format. Whisper models use these.

**GPU (Graphics Processing Unit)** The graphics card. Can run AI models 10-50x faster than CPU because of parallel processing.

---

**Greedy Decoding** A simple AI text generation strategy: always pick the most likely next word. Fast but not always optimal.

---

## H

**Hz (Hertz)** Samples per second. Human speech needs at least 8000 Hz to be understandable. Whisper uses 16000 Hz.

**hound** Rust crate for reading and writing WAV audio files.

---

## I

**Inference** Running a trained AI model on new data. "Running inference" = using the model to transcribe audio.

**invoke()** Tauri's function for calling Rust code from JavaScript. Returns a Promise.

**IPC (Inter-Process Communication)** How the frontend (webview) and backend (Rust) talk to each other in Tauri.

---

## L

**Lifetime** Rust's way of tracking how long references are valid. Written as '`a`'. The compiler uses these to prevent dangling references.

**LLM (Large Language Model)** AI models trained on text that can generate or modify text. Gemma is the LLM used for grammar correction.

---

## M

**Mono** Single-channel audio. Speech recognition requires mono (not stereo).

**Mutex (Mutual Exclusion)** A lock that ensures only one thread can access data at a time. Prevents data races.

---

## N

**Nemotron** NVIDIA's streaming ASR model. Can transcribe in real-time as audio comes in, unlike batch models.

**Nyquist Theorem** To capture a frequency, you need at least 2x the sample rate. Human speech is ~4000 Hz, so 8000+ Hz is needed. 16000 Hz gives headroom.

---

## O

**ONNX (Open Neural Network Exchange)** A standard format for AI models. Parakeet uses ONNX models.

**Option** Rust's way of representing "maybe a value, maybe nothing". `Some(value)` or `None`.

---

**Ownership** Rust's core concept: every value has exactly one owner. When the owner goes out of scope, the value is dropped.

---

## P

**Parakeet** NVIDIA's ASR engine. Includes several model types: Nemotron (streaming), CTC (batch), TDT (batch).

---

## Q

**Quantization** Compressing AI models by reducing number precision. Q5_1 means 5-bit quantization. Smaller but slightly less accurate.

---

## R

**Resampling** Converting audio from one sample rate to another. 48000 Hz → 16000 Hz. Uses interpolation to maintain quality.

**Result<T, E>** Rust's way of representing "either success or failure". `Ok(value)` or `Err(error)`.

**RMS (Root Mean Square)** A way to measure audio loudness. Square all samples, average them, take square root.

**rubato** Rust crate for high-quality audio resampling using sinc interpolation.

---

## S

**Sample** A single measurement of audio amplitude at a point in time. Audio is a stream of samples.

**Sample Rate** How many samples per second. 48000 Hz means 48,000 samples every second.

**serde** Rust crate for **serializing** and **deserializing** data. Converts Rust structs to/from JSON.

**Sinc Interpolation** A high-quality resampling algorithm that uses the sinc function. Better than simple linear interpolation.

**Spawn** Creating a new thread. `std::thread::spawn(|| { ... })`.

**State (Tauri)** Shared data that persists across command calls. Registered with `.manage()` and accessed via `State<T>`.

**Stereo** Two-channel audio (left and right). Must be converted to mono for speech recognition.

---

## T

**Tauri** Framework for building desktop apps with web frontend (HTML/JS/CSS) and Rust backend.

**TDT (Token-and-Duration Transducer)** A Parakeet model architecture that predicts both tokens and their durations.

**Thread** An independent execution path in a program. Multiple threads can run code simultaneously.

**Trait** Rust's version of interfaces. Defines behavior that types can implement. `Debug`, `Clone`, `Send` are traits.

**Transcript** The text output of speech recognition.

**tx / rx** Transmitter and Receiver ends of a channel. `tx.send()` puts data in, `rx.recv()` takes data out.

---

## U

**unsafe** Rust keyword that disables some safety checks. Required for certain operations like FFI or raw pointers.

**unwrap()** Extract the value from `Option` or `Result`. Panics if `None` or `Err`. Use carefully!

---

## V

**VAD (Voice Activity Detection)** Detecting whether audio contains speech or silence. Used to skip processing silent chunks.

**Vec** Rust's growable array type. `Vec<f32>` is a vector of floating-point numbers.

**Vulkan** Cross-platform GPU API. Works with AMD, Intel, and some NVIDIA GPUs. Used when CUDA isn't available.

---

## W

**WAV** Uncompressed audio file format. Stores raw samples. Easy to read/write but large files.

**Whisper** OpenAI's speech recognition model. Very accurate, supports many languages.

**whisper_rs** Rust bindings for whisper.cpp, which runs Whisper models efficiently.

---

## Quick Reference Table

Abbreviation	Full Name	What It Does
ASR	Automatic Speech Recognition	Speech → Text
VAD	Voice Activity Detection	Detect speech vs silence
CUDA	Compute Unified Device Architecture	NVIDIA GPU computing
ONNX	Open Neural Network Exchange	AI model format
GGUF	GPT-Generated Unified Format	Quantized model format
RMS	Root Mean Square	Audio loudness measure
IPC	Inter-Process Communication	Frontend ↔ Backend
LLM	Large Language Model	Text AI (Gemma)

Abbreviation	Full Name	What It Does
CTC	Connectionist Temporal Classification	ASR architecture
TDT	Token-and-Duration Transducer	ASR architecture
Hz	Hertz	Samples per second
kHz	Kilohertz	Thousands of samples/sec

## ⌚ Frontend Documentation (React + TypeScript)

Complete documentation of how the frontend makes calls to the Rust backend.

### Frontend File Structure

```
src/
├── main.tsx      # React entry point (10 lines)
├── App.tsx       # Main application component (667 lines)
├── App.css        # Styling (674 lines)
├── vite-env.d.ts  # Vite type declarations
└── assets/
    └── react.svg   # React logo
```

### main.tsx - Entry Point

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root") as HTMLElement).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
);
```

### What it does:

1. Imports React and ReactDOM
2. Finds the `#root` element in `index.html`
3. Renders the `<App />` component inside it
4. `StrictMode` enables additional development checks

### App.tsx - Main Component

#### TypeScript Interfaces (Lines 8-40)

These interfaces match the Rust structs exactly:

```
// Matches whisper::ModelInfo in Rust
interface ModelInfo {
    id: string;           // e.g., "tiny.en-q5_1"
    display_name: string; // e.g., "Tiny English (Q5_1)"
    file_name: string;    // e.g., "ggml-tiny.en-q5_1.bin"
    size_mb: number;      // e.g., 42.3
}

// Matches types::SampleFile in Rust
interface SampleFile {
    name: string;          // e.g., "test_audio.wav"
    path: string;           // e.g., "/full/path/to/test_audio.wav"
}

// Matches parakeet::ParakeetModelInfo in Rust
interface ParakeetModelInfo {
    id: string;           // e.g., "tdt:parakeet-tdt-0.6b-v2"
    display_name: string; // e.g., "Parakeet TDT - parakeet-tdt-0.6b-v2"
    model_type: string;   // e.g., "TDT", "CTC", "Nemotron"
    size_mb: number;
}

// Matches parakeet::ParakeetStatus in Rust
interface ParakeetStatus {
    loaded: boolean;
    model_id: string | null;
    model_type: string | null;
    backend: string;        // e.g., "CUDA", "DirectML", "CPU"
}

// Matches types::TranscriptionChunk in Rust
interface LiveTranscriptionPayload {
    text: string;
    processing_time_ms: number;
    method: string;         // "Whisper" or "Parakeet"
}

// Frontend-only type for engine selection
type ASREngine = "whisper" | "parakeet";
```

### Type Mapping (Rust → TypeScript):

Rust Type	TypeScript Type
String	string
f32, f64	number
u32, i32	number

Rust Type	TypeScript Type
-----------	-----------------

bool	boolean
------	---------

Option<T>	T \  null
-----------	-----------

Vec<T>	T[]
--------	-----

## State Variables (Lines 43-67)

```

function App() {
    // Persistent store reference (for saving settings)
    const storeRef = useRef<Store | null>(null);

    // UI State
    const [greetMsg, setGreetMsg] = useState(""); // Benchmark results
    const [liveTranscript, setLiveTranscript] = useState(""); // Real-time text
    const [latestLatency, setLatestLatency] = useState<number | null>(null);
    const [isRecording, setIsRecording] = useState(false);
    const [isLoading, setIsLoading] = useState(false);
    const [isInitialLoading, setIsInitialLoading] = useState(true);
    const [loadingMessage, setLoadingMessage] = useState("");

    // Backend State (synced from Rust)
    const [backendInfo, setBackendInfo] = useState("Loading..."); // "CUDA"/"CPU"
    const [models, setModels] = useState<ModelInfo[]>([]);
    const [currentModel, setCurrentModel] = useState<string | null>(null);
    const [sampleFiles, setSampleFiles] = useState<SampleFile[]>([]);
    const [selectedSample, setSelectedSample] = useState<string>("");

    // Parakeet-specific state
    const [parakeetModels, setParakeetModels] = useState<ParakeetModelInfo[]>([]);
    const [currentParakeetModel, setCurrentParakeetModel] = useState<string | null>(null);
    const [, setParakeetStatus] = useState<ParakeetStatus | null>(null);

    // Engine selection
    const [activeEngine, setActiveEngine] = useState<ASREngine>("whisper");

    // LLM (Grammar correction) state
    const [isCorrecting, setIsCorrecting] = useState(false);
    const [llmStatus, setLlmStatus] = useState("Not Loaded");
    const [enableGrammarLM, setEnableGrammarLM] = useState(false);

    // Refs for hotkey state (avoid stale closures in async callbacks)
    const isRecordingRef = useRef(false);
    const startingRecordingRef = useRef(false);
    const pendingStopRef = useRef(false);
}

```

## Why useRef for hotkey state?

### useState vs useRef FOR ASYNC CALLBACKS

PROBLEM: useState creates "stale closures" in event listeners

BAD: Using useState in event listener

```
const [isRecording, setIsRecording] = useState(false);

useEffect(() => {
  listen("hotkey-stop", () => {
    if (isRecording) { // ✗ STALE! Always sees initial value (false)
      stopRecording();
    }
  });
}, []); // Empty deps = listener created once with initial state
```

GOOD: Using useRef for mutable value

```
const isRecordingRef = useRef(false);

// Update ref whenever state changes
const [isRecording, setIsRecording] = useState(false);
isRecordingRef.current = isRecording;

useEffect(() => {
  listen("hotkey-stop", () => {
    if (isRecordingRef.current) { // ✓ Always sees latest value!
      stopRecording();
    }
  });
}, []);
```

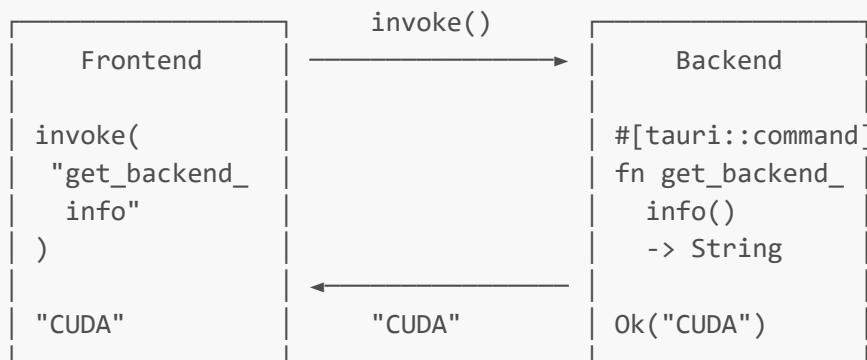
#### WHY IT WORKS:

- useState: Each render creates new value, closure captures ONE of them
- useRef: Same object across all renders, .current always up-to-date

## How invoke() Calls Work

### Pattern 1: Simple Call (Get Data)

```
// Get backend info (CUDA/Vulkan/CPU)
const backend = await invoke("get_backend_info");
setBackendInfo(backend as string);
```

**Flow:****Pattern 2: Call with Parameters**

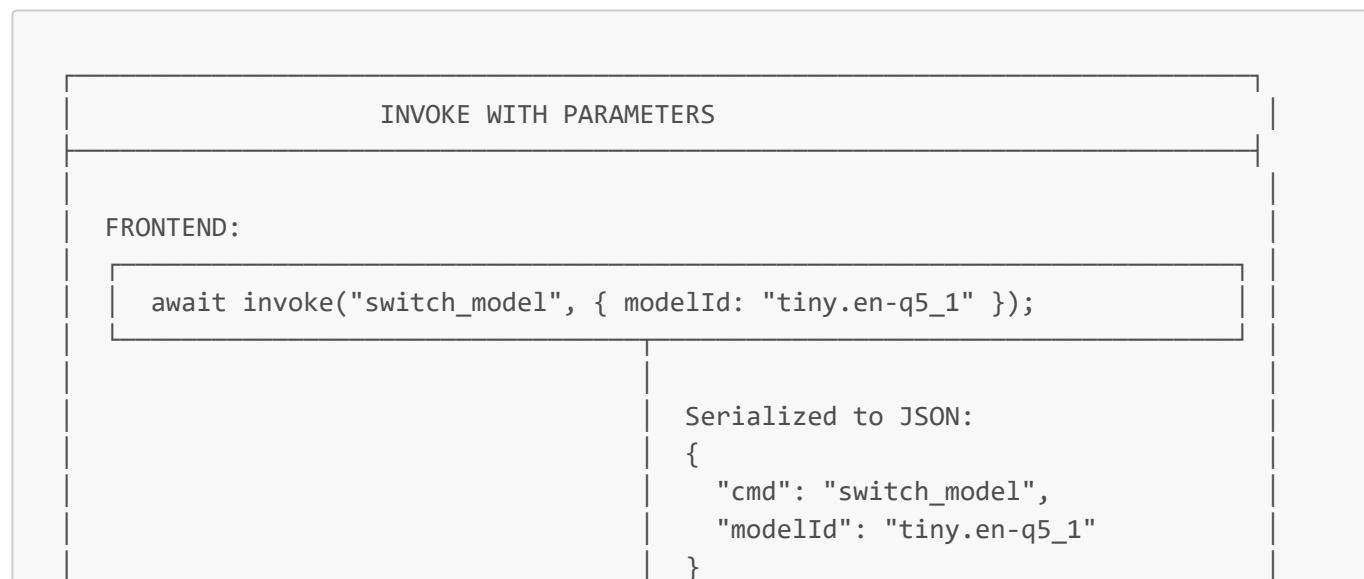
```

// Switch Whisper model
const result = await invoke("switch_model", { modelId: "large-v3-turbo-q5_1" });
//                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//                                         Object with named parameters
  
```

**Parameter Naming:**

TypeScript (camelCase) → Rust (snake_case)

{ modelId: "tiny.en" }	→	model_id: String
{ filePath: "/path" }	→	file_path: String
{ newState: "ready" }	→	new_state: String

**Flow:**

BACKEND:

```
#[tauri::command]
pub fn switch_model(
    state: State<AudioState>, // Injected by Tauri
    model_id: String,         // ← Deserialized from "modelId"
) -> Result<String, String>
```

Returns: Ok("Backend: CUDA")

FRONTEND RECEIVES:

```
const result = "Backend: CUDA";
toast.success(`☑ ${result}`);
```

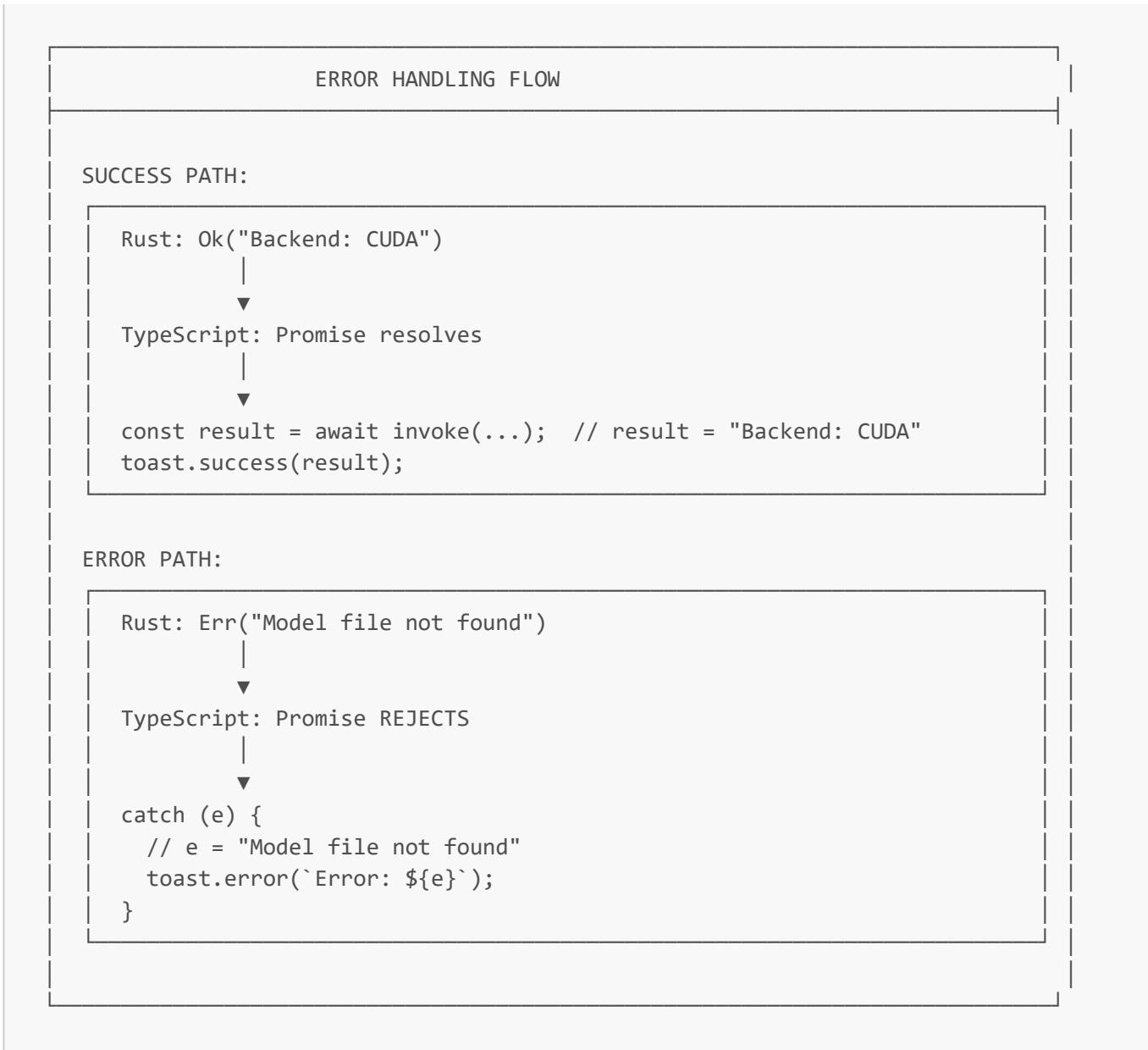
### Pattern 3: Call with Error Handling

```
const handleModelChange = async (modelId: string) => {
    setIsLoading(true);
    setLoadingMessage(`Loading ${modelId}...`);

    try {
        await setTrayState("processing"); // Update tray icon
        const result = await invoke("switch_model", { modelId });
        setCurrentModel(modelId);
        toast.success(`☑ ${result}`);

        // Refresh related state
        const backend = await invoke("get_backend_info");
        setBackendInfo(backend as string);
    } catch (e) {
        toast.error(`☒ Error switching model: ${e}`); // e is the Err string
    } finally {
        setIsLoading(false);
        setLoadingMessage("");
        await setTrayState("ready");
    }
};
```

### Error Flow:



---

## How listen() Events Work

### Setting Up Event Listeners

```
useEffect(() => {
  let unlistenStart: (() => void) | undefined;
  let unlistenStop: (() => void) | undefined;
  let unlistenChunk: (() => void) | undefined;

  const setupListeners = async () => {
    // Listen for hotkey events from Rust
    unlistenStart = await listen("hotkey-start-recording", async () => {
      console.log("[HOTKEY] Ctrl+Win pressed - starting recording");
      await handleStartRecording();
    });

    unlistenStop = await listen("hotkey-stop-recording", async () => {
      console.log("[HOTKEY] Ctrl+Win released - stopping recording");
    });
  }
});
```

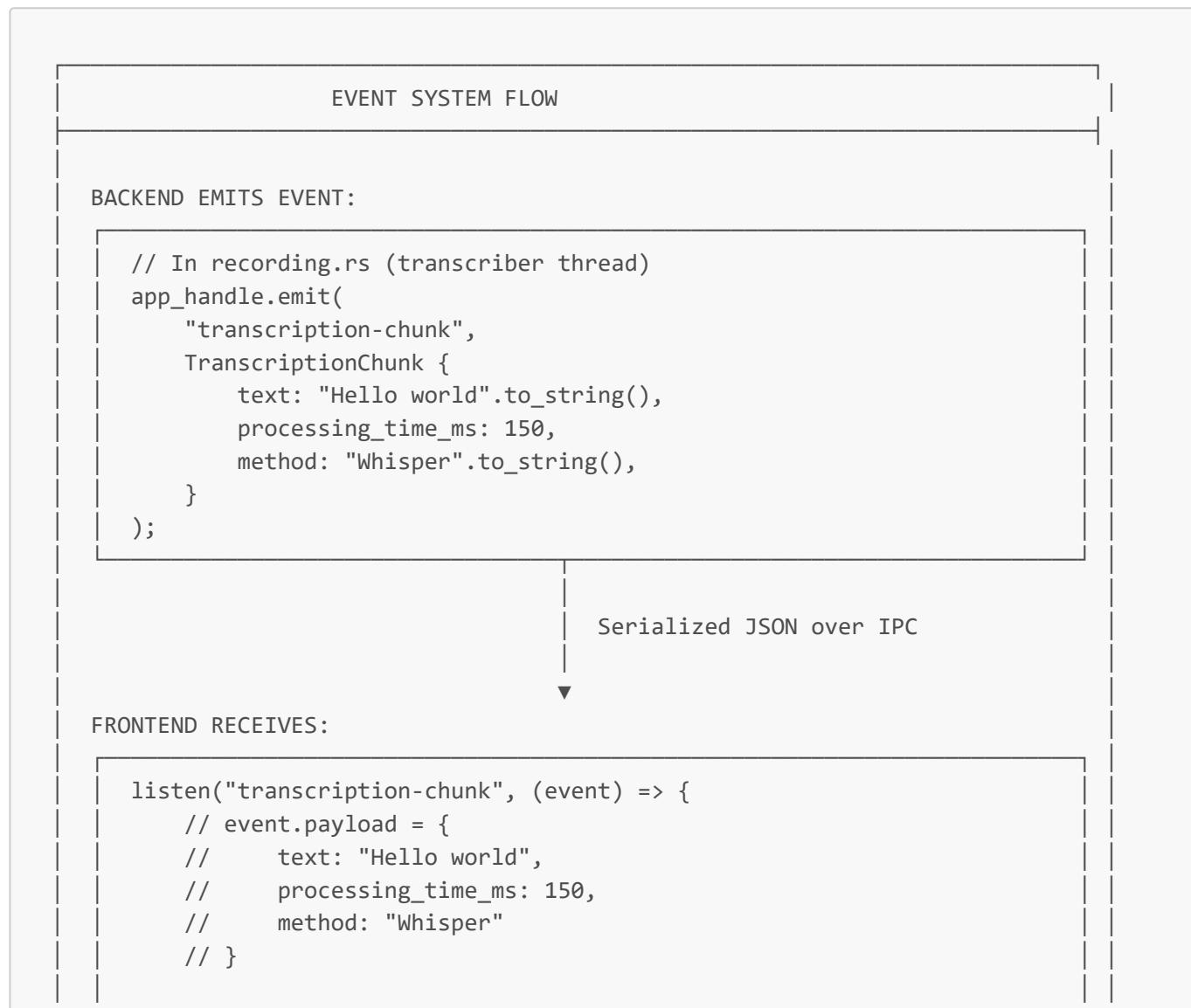
```
    await handleStopRecording();
});

// Listen for real-time transcription chunks
unlistenChunk = await listen("transcription-chunk", (event) => {
  const payload = event.payload as LiveTranscriptionPayload;
  setLiveTranscript((prev) => prev + (prev ? " " : "") + payload.text);
  setLatestLatency(payload.processing_time_ms);
});
};

setupListeners();

// Cleanup: Remove listeners when component unmounts
return () => {
  if (unlistenStart) unlistenStart();
  if (unlistenStop) unlistenStop();
  if (unlistenChunk) unlistenChunk();
};
}, []); // Empty deps = run once on mount
```

## Event Flow:

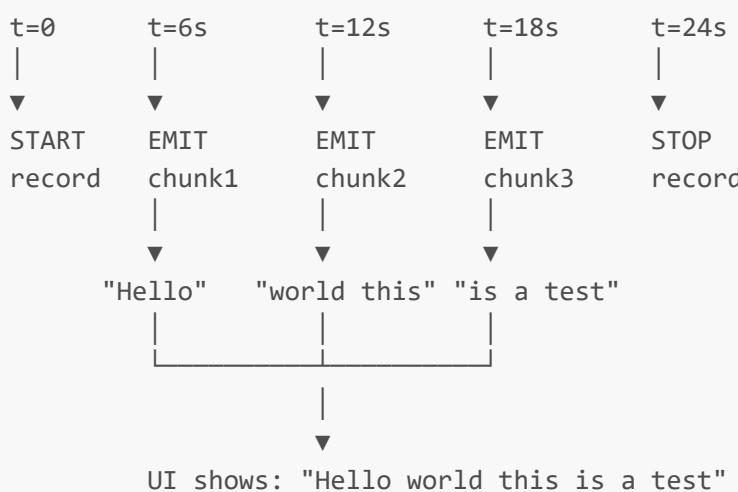


```

    const payload = event.payload as LiveTranscriptionPayload;
    setLiveTranscript(prev => prev + " " + payload.text);
    setLatestLatency(payload.processing_time_ms);
  });
}

```

## TIMELINE:



## All Frontend → Backend Calls

Location	invoke() Call	When Triggered
<b>Initial Load (useEffect)</b>		
Line 74	invoke("get_backend_info")	Component mounts
Line 78	invoke("list_models")	Component mounts
Line 82	invoke("get_current_model")	Component mounts
Line 86	invoke("list_sample_files")	Component mounts
Line 95	invoke("list_parakeet_models")	Component mounts
Line 98	invoke("get_parakeet_status")	Component mounts
<b>Engine Sync (useEffect)</b>		
Line 143	invoke("set_active_engine", { engine })	Engine changes
<b>Recording Controls</b>		
Line 168	invoke("start_recording")	Start button / hotkey
Line 189	invoke("stop_recording")	Stop button / hotkey
Line 197	invoke("correct_text", { text })	After stop (if LLM enabled)

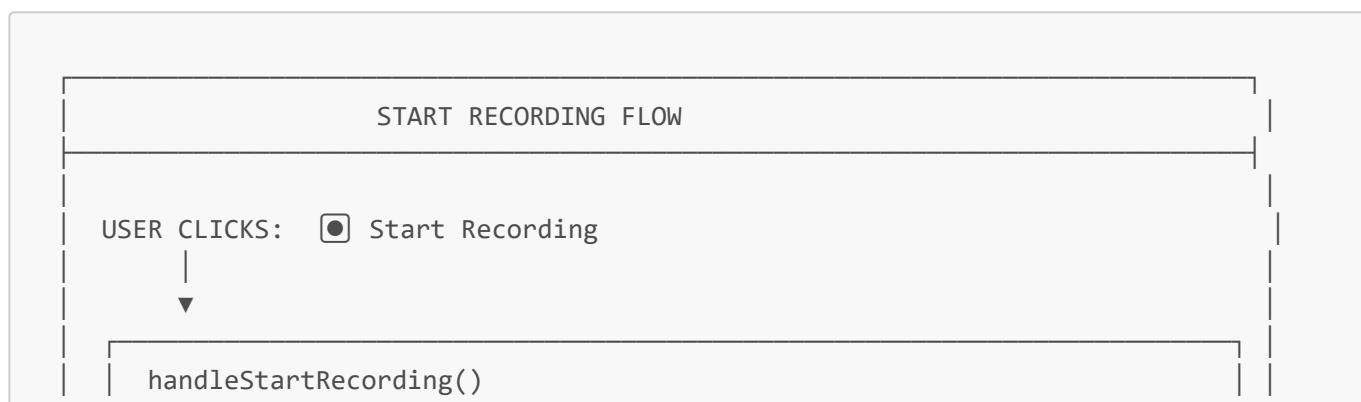
Location	invoke() Call	When Triggered
<b>Model Switching</b>		
Line 316	invoke("switch_model", { modelId })	Whisper dropdown change
Line 328	invoke("get_backend_info")	After model switch
Line 347	invoke("init_parakeet", { modelId })	Parakeet dropdown change
Line 358	invoke("get_parakeet_status")	After Parakeet init
<b>Tray Icon</b>		
Line 404	invoke("set_tray_state", { newState })	State changes
<b>LLM</b>		
Line 434	invoke("init_llm")	Toggle LLM on
Line 591	invoke("init_llm")	Load LLM button
Line 642	invoke("correct_text", { text })	Correct Grammar button
<b>Benchmark</b>		
Line 569	invoke("benchmark_test", { filePath })	Run Benchmark button

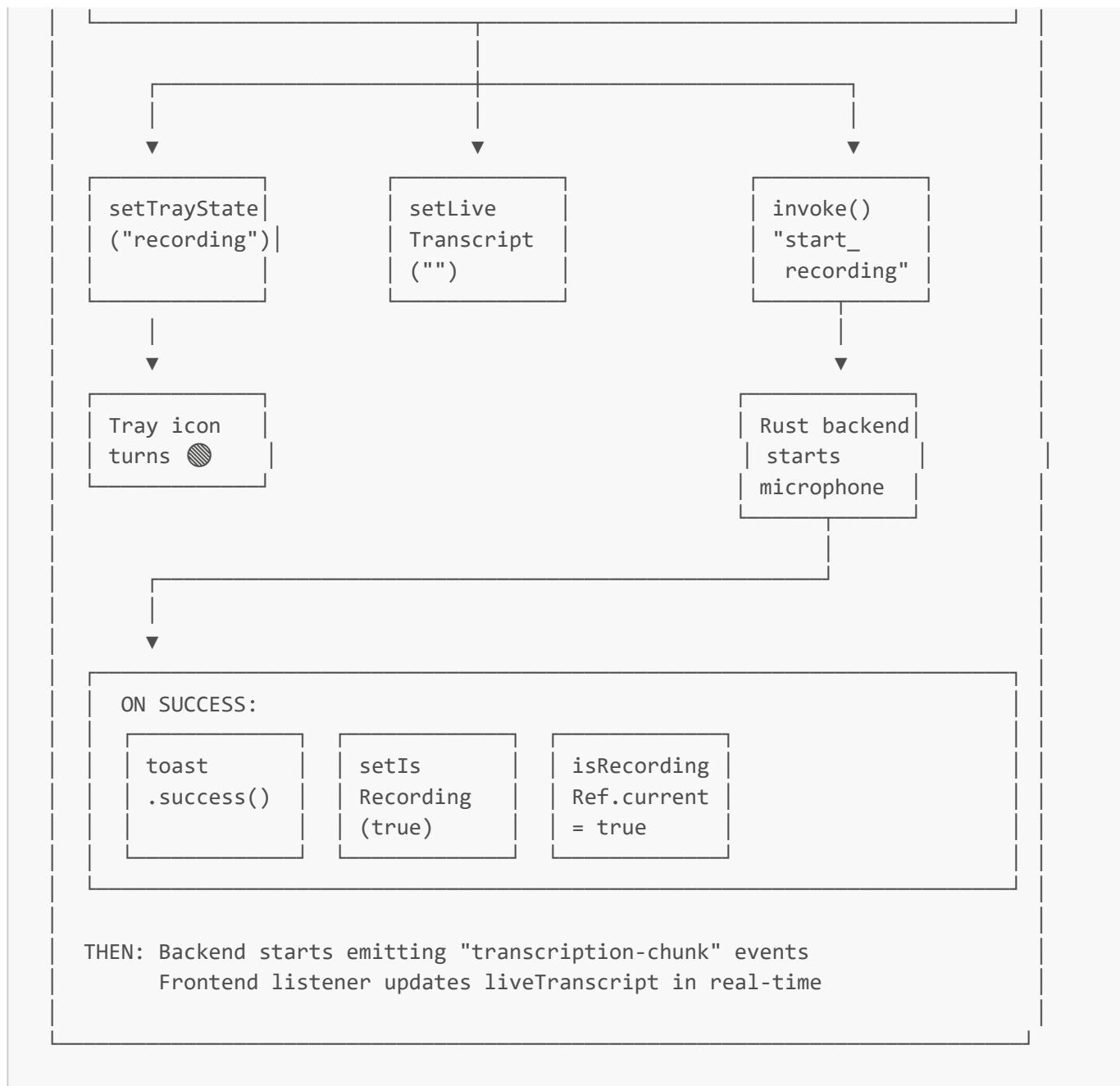
All Backend → Frontend Events

Event Name	Emitted From	Payload	Handler
hotkey-start-recording	hotkeys/listener.rs	None	Calls handleStartRecording()
hotkey-stop-recording	hotkeys/listener.rs	None	Calls handleStopRecording()
transcription-chunk	commands/recording.rs	LiveTranscriptionPayload	Updates liveTranscript state

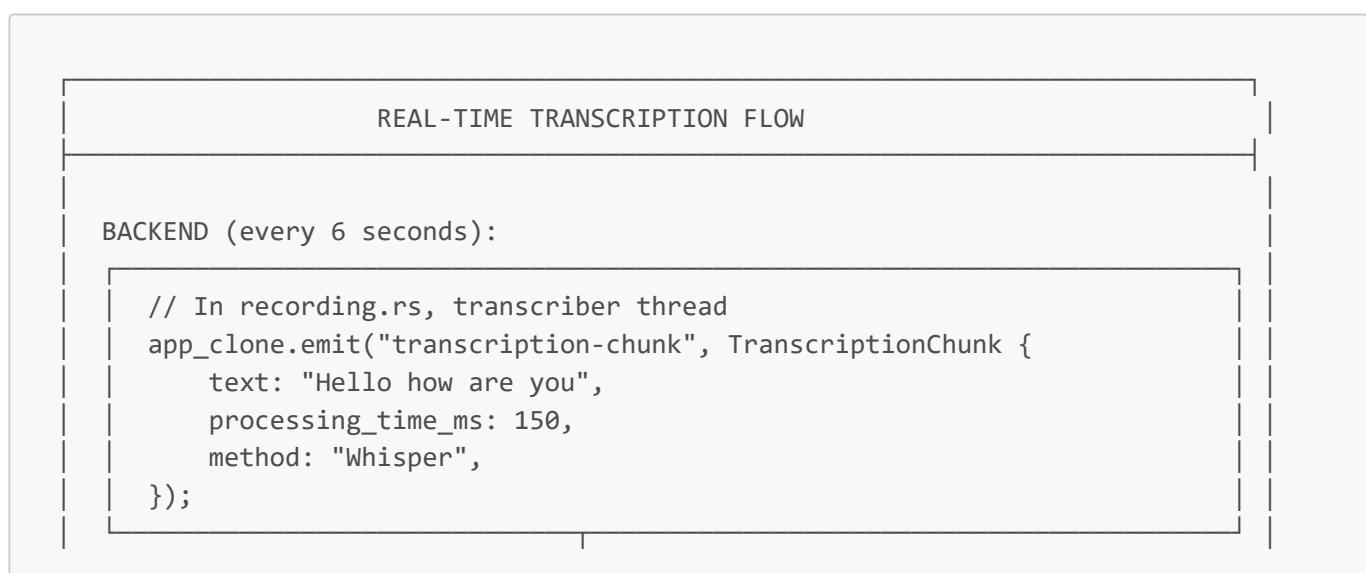
User Action Flow Diagrams

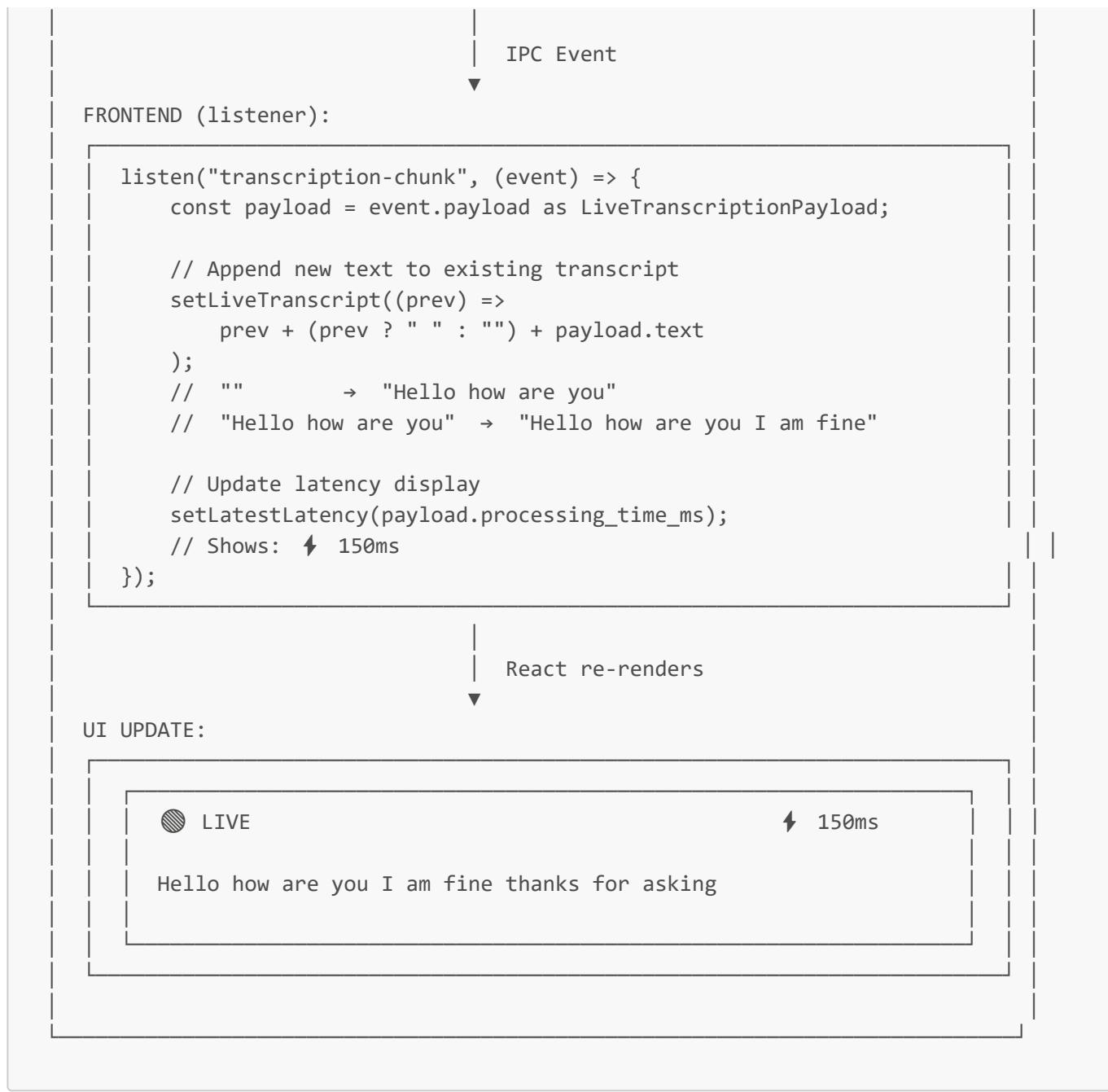
#### Flow 1: Click "Start Recording"



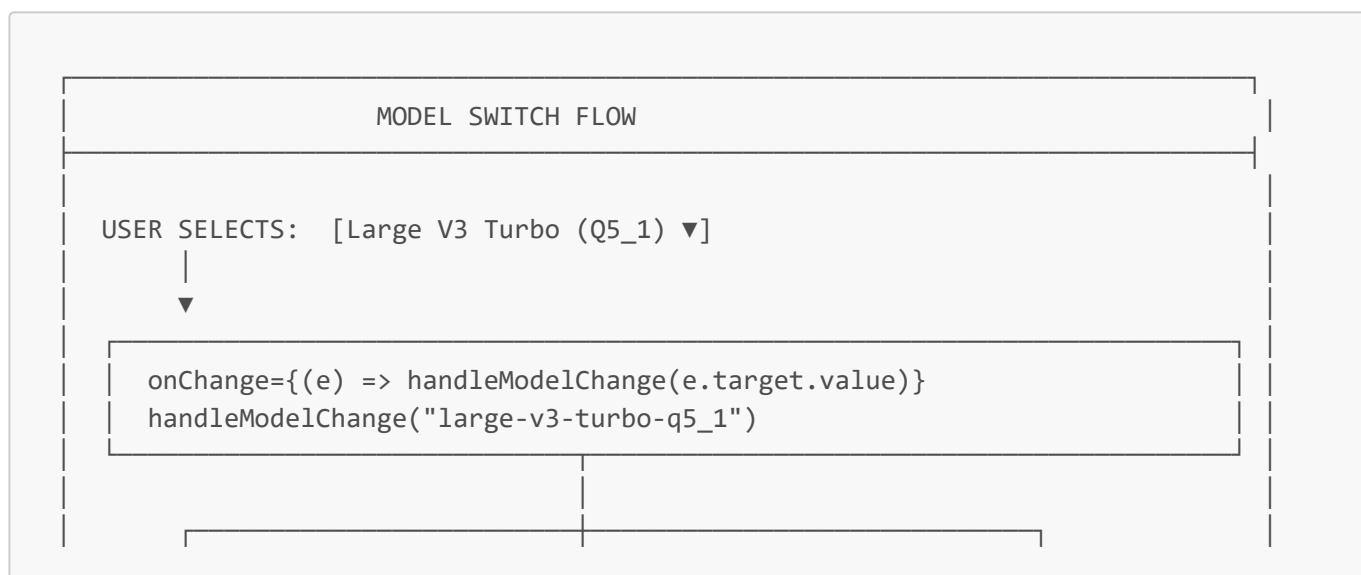


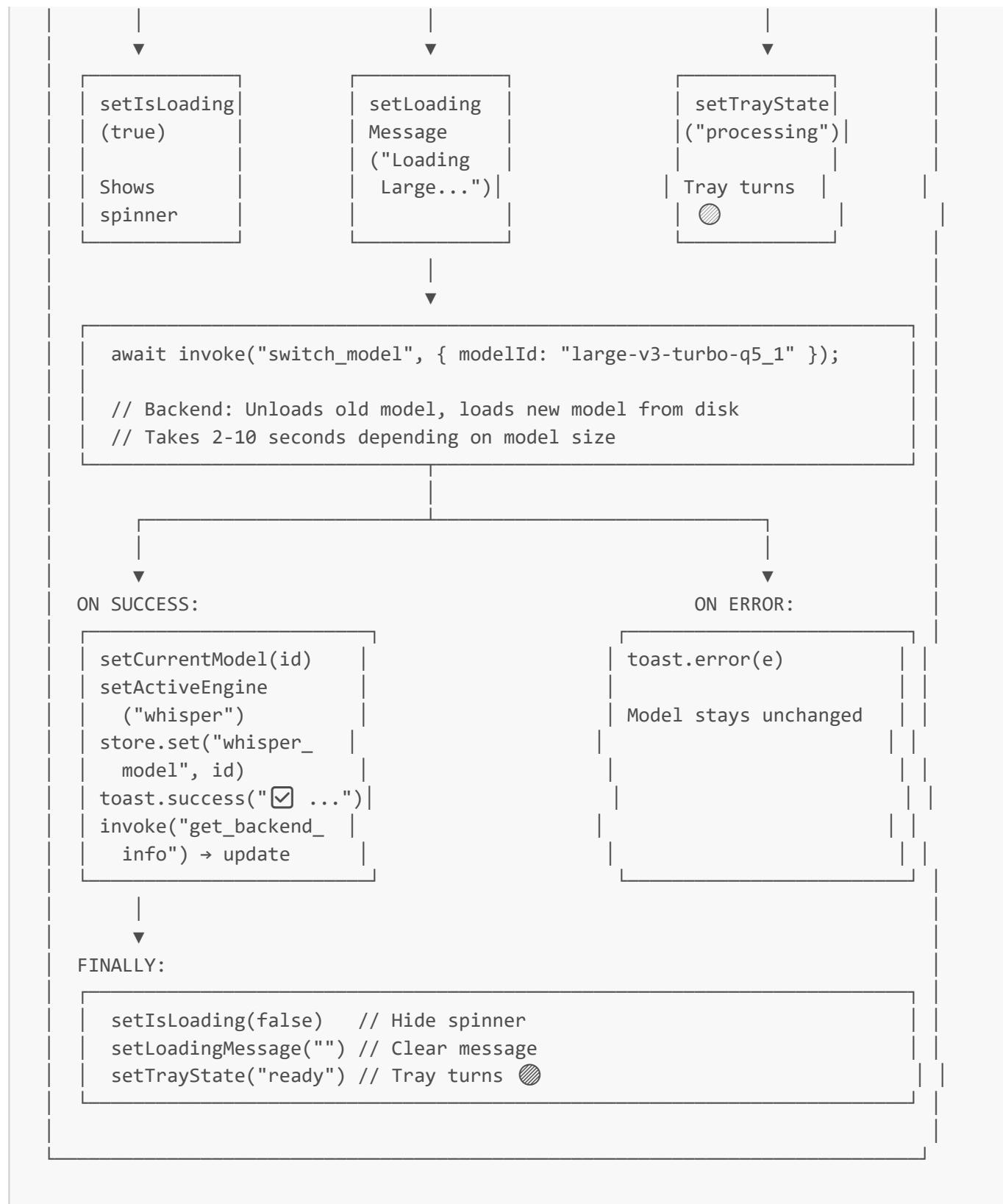
## Flow 2: Real-Time Transcription Update



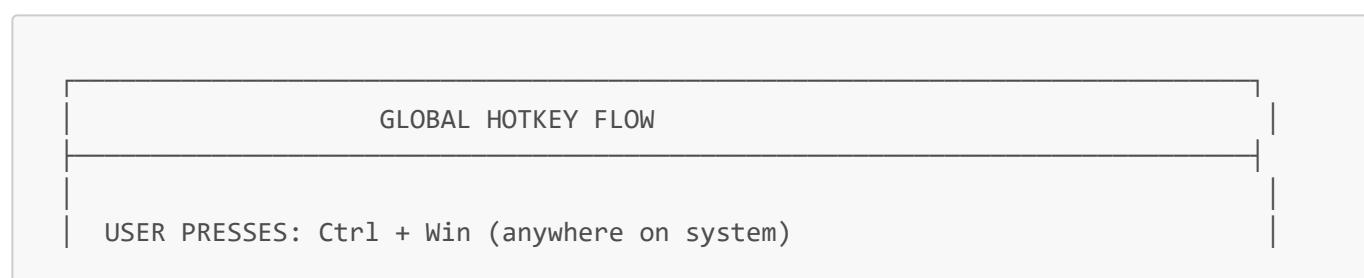


### Flow 3: Switch Model





#### Flow 4: Hotkey (Ctrl+Win)



```
RUST: hotkeys/listener.rs
```

```
rdev::listen(|event| {
    if ctrl_held && meta_held && !recording_active {
        recording_active = true;
        app_handle.emit("hotkey-start-recording", ());
    }
});
```

Event emitted

```
FRONTEND: App.tsx listener
```

```
listen("hotkey-start-recording", async () => {
    // Debounce check
    if (Date.now() - lastStartTime < 500) return;

    // State check via ref (not stale state)
    if (!isRecordingRef.current && !startingRecordingRef.current) {
        startingRecordingRef.current = true;
        await handleStartRecording(); // Same as button click!
        startingRecordingRef.current = false;
    }
});
```

USER RELEASES: Ctrl or Win

```
RUST: hotkeys/listener.rs
```

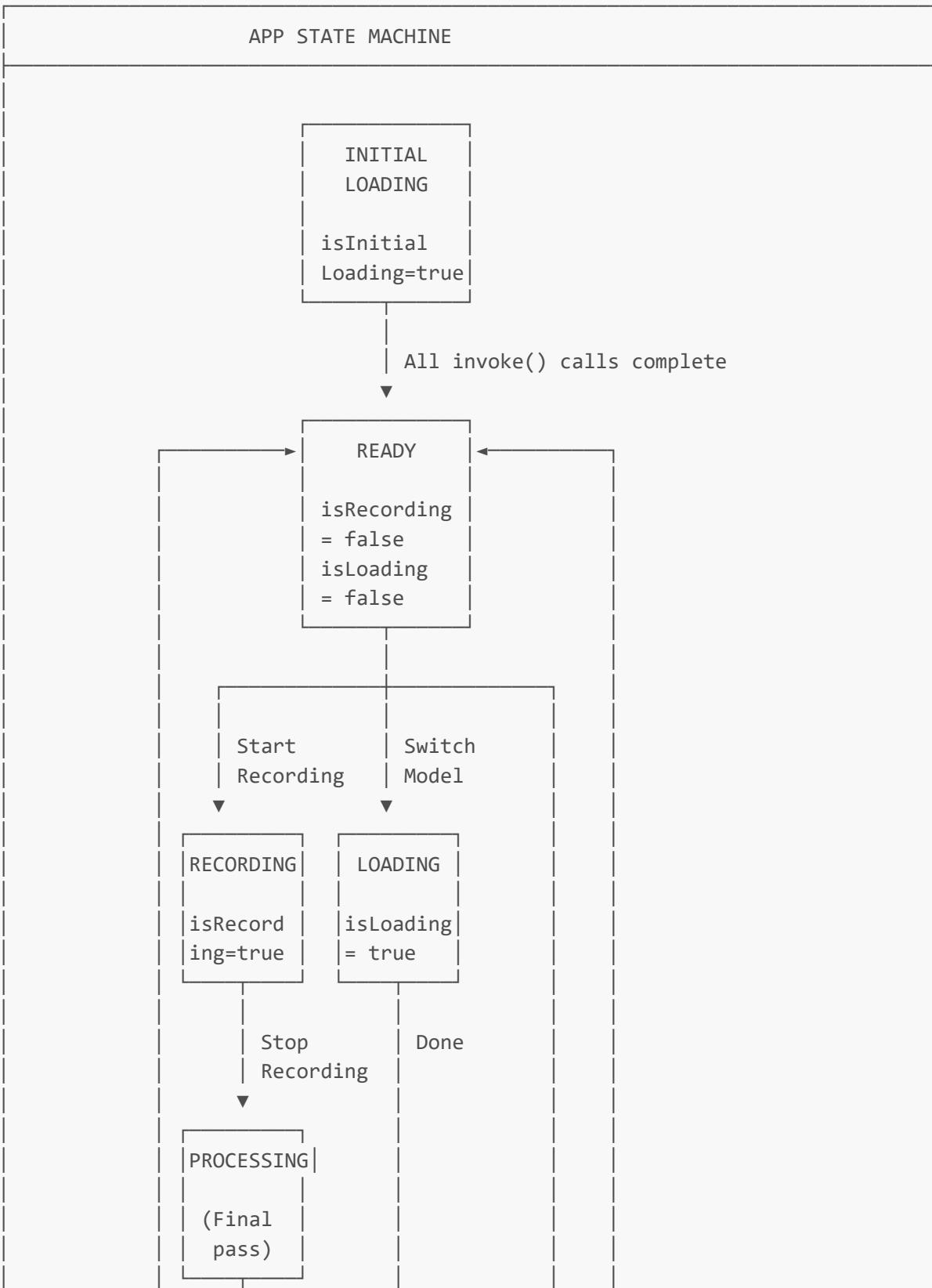
```
if recording_active && (!ctrl_held || !meta_held) {
    recording_active = false;
    app_handle.emit("hotkey-stop-recording", ());
}
```

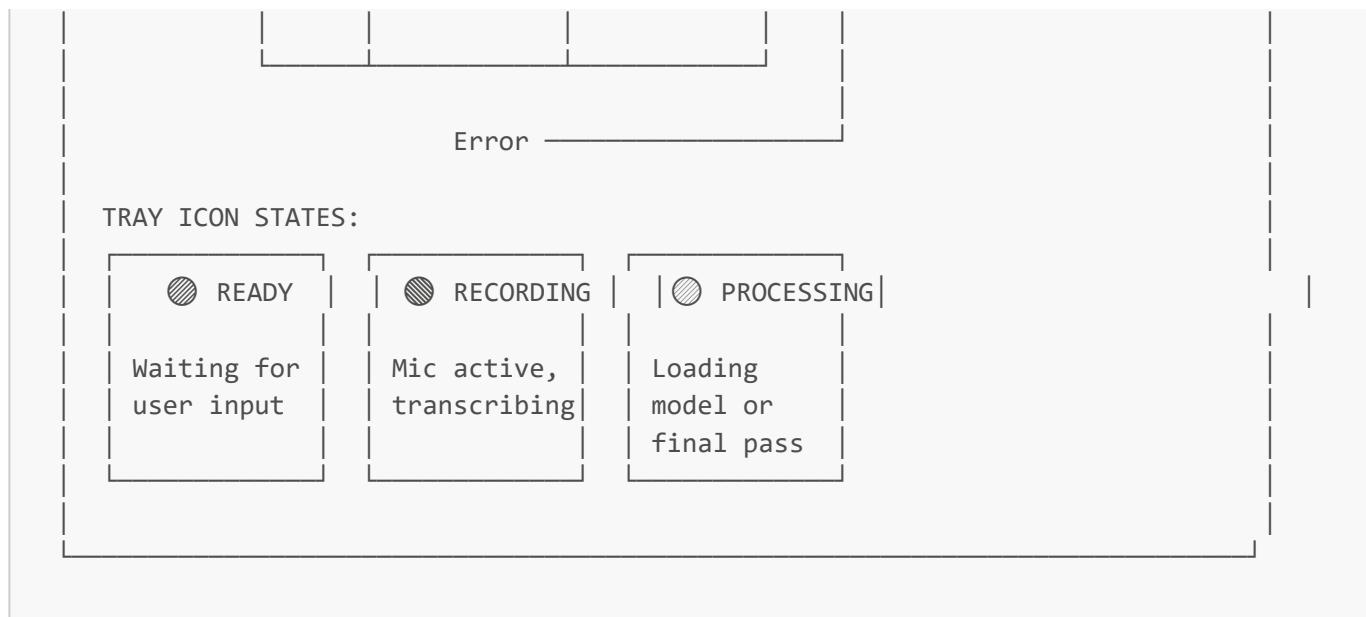
```
FRONTEND: listener
```

```
listen("hotkey-stop-recording", async () => {
    if (isRecordingRef.current) {
        await handleStopRecording(); // Same as button click!
    }
});
```

RESULT: Hold Ctrl+Win to record, release to stop.  
Works from ANY application on the system!

## Frontend State Machine





## CSS Architecture

```
App.css Structure:  
├── :root (CSS Variables)  
│   ├── Color palette (--bg-primary, --accent-primary, etc.)  
│   ├── Typography settings  
│   └── Common values (--border-color, --shadow-color)  
├── Base styles (body, *, .container)  
├── Component styles  
│   ├── .status-bar-container, .status-card (Engine cards)  
│   ├── .engine-badge (Whisper/Parakeet labels)  
│   ├── .model-section, .model-select (Dropdown)  
│   ├── .controls, .btn, .btn-start, .btn-stop (Buttons)  
│   ├── .output-area, .live-transcript (Results)  
│   ├── .loading-overlay, .loading-spinner (Loading state)  
│   └── .switch, .slider (Toggle switch)  
├── Animations (@keyframes)  
│   ├── spin (Loading spinner)  
│   ├── fadeIn (Component entrance)  
│   └── pulse (Live indicator)  
└── Responsive design (@media queries)
```

## Quick Reference: Frontend → Backend

FRONTEND-BACKEND CHEAT SHEET

## CALL BACKEND FUNCTION:

```

import { invoke } from "@tauri-apps/api/core";

// No parameters
const result = await invoke("command_name");

// With parameters (camelCase → snake_case automatic)
const result = await invoke("command_name", { paramName: value });

// With error handling
try {
    const result = await invoke("command_name");
} catch (error) {
    console.error(error); // error is the Err(...) string from Rust
}

```

## LISTEN FOR BACKEND EVENTS:

```

import { listen } from "@tauri-apps/api/event";

useEffect(() => {
    let unlisten: (() => void) | undefined;

    const setup = async () => {
        unlisten = await listen("event-name", (event) => {
            const payload = event.payload as MyType;
            // Handle event...
        });
    };
    setup();

    return () => { if (unlisten) unlisten(); }; // Cleanup!
}, []);

```

## TYPE MAPPING:

Rust	TypeScript
String	string
i32, u32, f32, f64	number
bool	boolean
Option<T>	T   null
Vec<T>	T[]
Result<T, E>	T (or throws E)
()	void / null
Field names: snake_case → camelCase	
processing_time_ms → processingTimeMs	

## 📦 Rust Crates & Dependencies Guide

Complete documentation of every crate used in this project, what it does, and how it's used.

### Cargo.toml Structure Overview

```
CARGO.TOML DEPENDENCY STRUCTURE

[build-dependencies]
└─ tauri-build          # Used during compilation
    # Compiles Tauri app resources

[dependencies]
└─ tauri                  # Used on ALL platforms
    # App framework (window, IPC, tray)
    └─ serde/serde_json      # JSON serialization
    └─ cpal                  # Audio capture
    └─ crossbeam-channel     # Thread communication
    └─ hound                 # WAV file I/O
    └─ chrono                # Date/time handling
    └─ rubato                # Audio resampling
    └─ dirs                  # OS directories (AppData, etc.)
    └─ rdev                  # Global hotkeys
    └─ regex                 # Text pattern matching
    └─ candle-*              # ML framework (Gemma LLM)
    └─ tokenizers             # LLM tokenization
    └─ anyhow                # Error handling

[target.'cfg(...).dependencies] # Platform-specific
└─ whisper-rs                 # Speech-to-text (Whisper)
    └─ parakeet-rs            # Speech-to-text (Parakeet)
        └─ ort                  # ONNX Runtime
```

### Core Framework Crates

#### 1. **tauri** - Application Framework

```
tauri = { version = "2", features = ["tray-icon", "image-png"] }
```

**What it is:** The core Tauri framework that creates desktop applications with a web frontend and Rust backend.

## Features enabled:

- `tray-icon`: System tray support (the colored circles)
- `image-png`: PNG image loading for tray icons

## How it's used:

```
// In lib.rs - App builder
tauri::Builder::default()
    .plugin(tauri_plugin_opener::init())      // Plugin registration
    .plugin(tauri_plugin_store::Builder::default().build())
    .plugin(tauri_plugin_fs::init())
    .manage(audio_state)                      // State management
    .invoke_handler(tauri::generate_handler![...]) // Command registration
    .setup(|app| { ... })                    // App initialization
    .run(tauri::generate_context!())          // Run the app

// In commands - Accessing Tauri features
use tauri::{AppHandle, State, Emitter, Manager};

#[tauri::command]
pub fn my_command(
    app: AppHandle,                  // Access to app methods
    state: State<AudioState>,       // Managed state
) -> Result<String, String> {
    app.emit("event-name", payload)?; // Emit to frontend
    Ok("Success".to_string())
}
```

**Analogy:** Tauri is like the **skeleton and nervous system** of your app - it provides the structure (windows, menus) and communication pathways (IPC between frontend and backend).

## 2. `tauri-plugin-store` - Persistent Settings

```
tauri-plugin-store = "2.4.2"
```

**What it is:** A plugin for persisting user settings as JSON files.

## How it's used:

```
// In lib.rs - Register plugin
.plugin(tauri_plugin_store::Builder::default().build())
```

```
// In App.tsx - Frontend usage
import { Store } from "@tauri-apps/plugin-store";
```

```
const store = await Store.load("settings.json");
await store.set("active_engine", "whisper");
await store.save();

const engine = await store.get<string>("active_engine");
```

**Where data is stored:** %APPDATA%/com.taurscribe.app/settings.json (Windows)

---

### 3. tauri-plugin-fs - File System Access

```
tauri-plugin-fs = "2"
```

**What it is:** Provides access to file system operations with security permissions.

**How it's used:** Allows the app to read/write files in allowed directories (AppData, etc.).

---

### 4. tauri-plugin-opener - External Links

```
tauri-plugin-opener = "2"
```

**What it is:** Opens URLs in the default browser or files with their associated applications.

---

## Serialization Crates

### 5. serde - Serialization Framework

```
serde = { version = "1", features = ["derive"] }
```

**What it is:** The de-facto Rust serialization framework. Converts Rust structs to/from other formats (JSON, TOML, etc.).

**The derive feature:** Auto-generates serialization code via #[derive(Serialize, Deserialize)].

**How it's used:**

```
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
pub struct ModelInfo {
    pub id: String,
    pub display_name: String,
```

```
    pub size_mb: f64,  
}  
// Serde auto-generates the code to convert this to/from JSON:  
// { "id": "tiny.en", "display_name": "Tiny English", "size_mb": 42.3 }
```

**Why it matters:** Every struct that crosses the frontend-backend boundary MUST be Serialize/Deserialize.

---

## 6. **serde_json - JSON Handling**

```
serde_json = "1"
```

**What it is:** JSON-specific serialization using Serde.

**How it's used:**

```
use serde_json;  
  
let json_string = serde_json::to_string(&my_struct)?;  
let my_struct: MyStruct = serde_json::from_str(&json_string)?;
```

---

Audio Processing Crates

## 7. **cpal - Cross-Platform Audio Library**

```
cpal = "0.15"
```

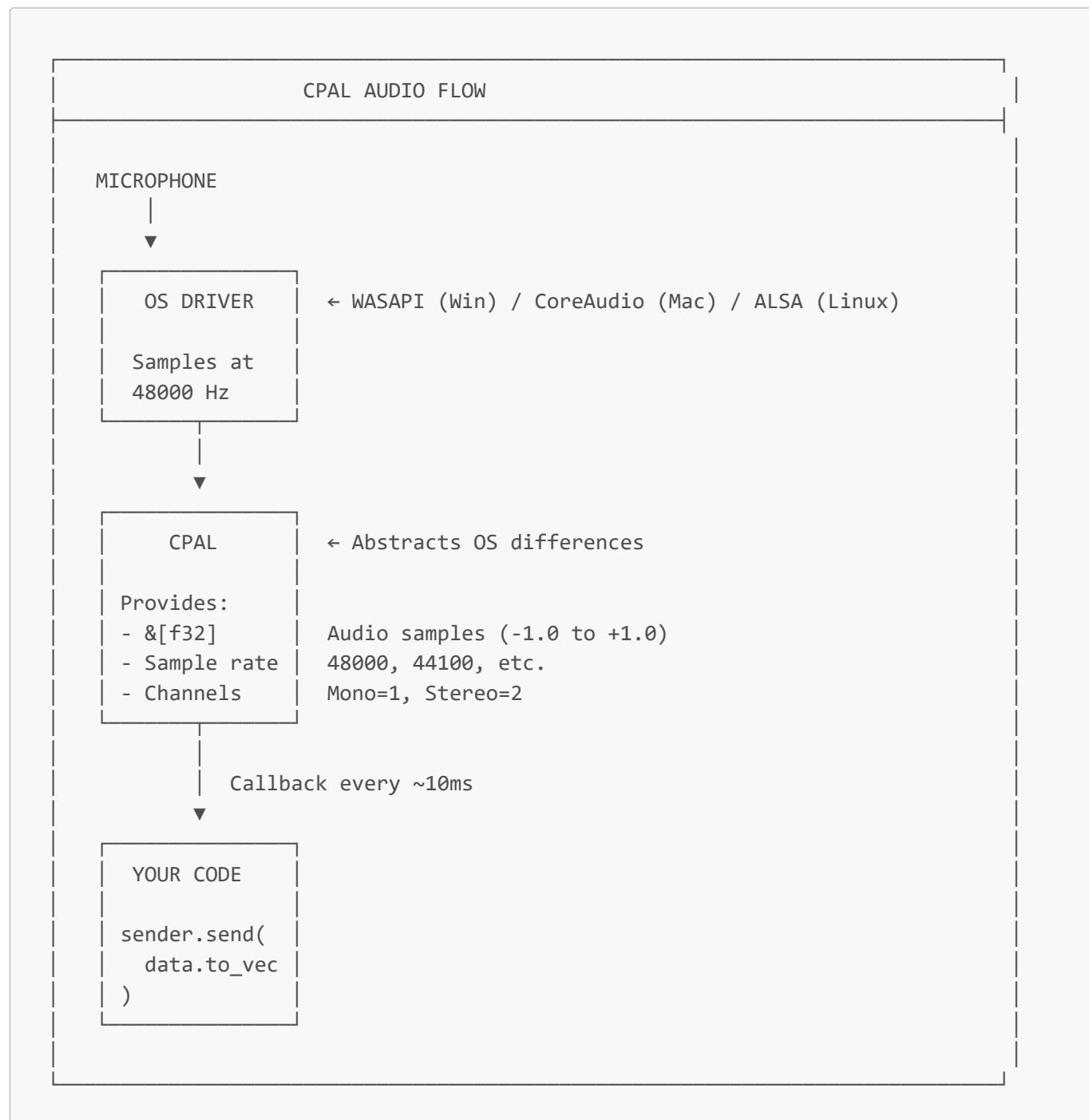
**What it is:** Low-level cross-platform audio I/O. Interfaces with OS audio APIs (WASAPI on Windows, CoreAudio on macOS, ALSA on Linux).

**How it's used:**

```
use cpal::traits::{DeviceTrait, HostTrait, StreamTrait};  
  
// Get the default audio host (WASAPI/CoreAudio/ALSA)  
let host = cpal::default_host();  
  
// Get the default input device (microphone)  
let device = host.default_input_device()  
    .ok_or("No input device found")?  
  
// Get supported audio format  
let config = device.default_input_config()?  
// e.g., 48000 Hz, 2 channels, f32 samples
```

```
// Build audio stream with callback
let stream = device.build_input_stream(
    &config.into(),
    move |data: &[f32], _: &cpal::InputCallbackInfo| {
        // This runs every ~10ms with new audio samples
        sender.send(data.to_vec()).ok();
    },
    |err| eprintln!("Stream error: {}", err),
    None,
)?;

stream.play()?; // Start capturing
```

**Diagram:**

## 8. crossbeam-channel - Thread Communication

```
crossbeam-channel = "0.5"
```

**What it is:** High-performance multi-producer multi-consumer channels for thread communication. Faster and more feature-rich than `std::sync::mpsc`.

**How it's used:**

```
use crossbeam_channel::{unbounded, Sender, Receiver};

// Create channel pair
let (sender, receiver): (Sender<Vec<f32>>, Receiver<Vec<f32>>) = unbounded();

// Audio thread sends samples
std::thread::spawn(move || {
    let samples = vec![0.1, 0.2, 0.3];
    sender.send(samples).ok();
});

// Processing thread receives samples
std::thread::spawn(move || {
    while let Ok(samples) = receiver.recv() {
        // Process audio...
    }
});
```

**Why crossbeam over std::sync::mpsc?**

Feature	std::sync::mpsc	crossbeam-channel
Multi-producer	✓	✓
Multi-consumer	✗	✓
Select (wait on multiple)	✗	✓
Bounded channels	✗	✓
Performance	Good	Better

## 9. hound - WAV File I/O

```
hound = "3.5"
```

**What it is:** Read and write WAV audio files.

**How it's used:**

```
use hound::{WavSpec, WavWriter, SampleFormat};

// Create WAV file
let spec = WavSpec {
    channels: 1,           // Mono
    sample_rate: 16000,     // 16kHz (Whisper's expected rate)
    bits_per_sample: 32,   // 32-bit float
    sample_format: SampleFormat::Float,
};

let mut writer = WavWriter::create(path, spec)?;

// Write samples
for sample in audio_samples {
    writer.write_sample(sample)?;
}
writer.finalize()?; // Important! Writes header with correct length
```

**Reading WAV:**

```
use hound::WavReader;

let mut reader = WavReader::open(path)?;
let samples: Vec<f32> = reader.samples::<f32>()
    .filter_map(Result::ok)
    .collect();
```

**10. rubato - Audio Resampling**

```
rubato = "0.14"
```

**What it is:** High-quality audio sample rate conversion. Converts between sample rates (e.g., 48000Hz → 16000Hz) while preserving audio quality.

**Why it's needed:** Whisper requires exactly 16000Hz audio. Microphones typically provide 44100Hz or 48000Hz.

**How it's used:**

```
use rubato::{FftFixedIn, Resampler};

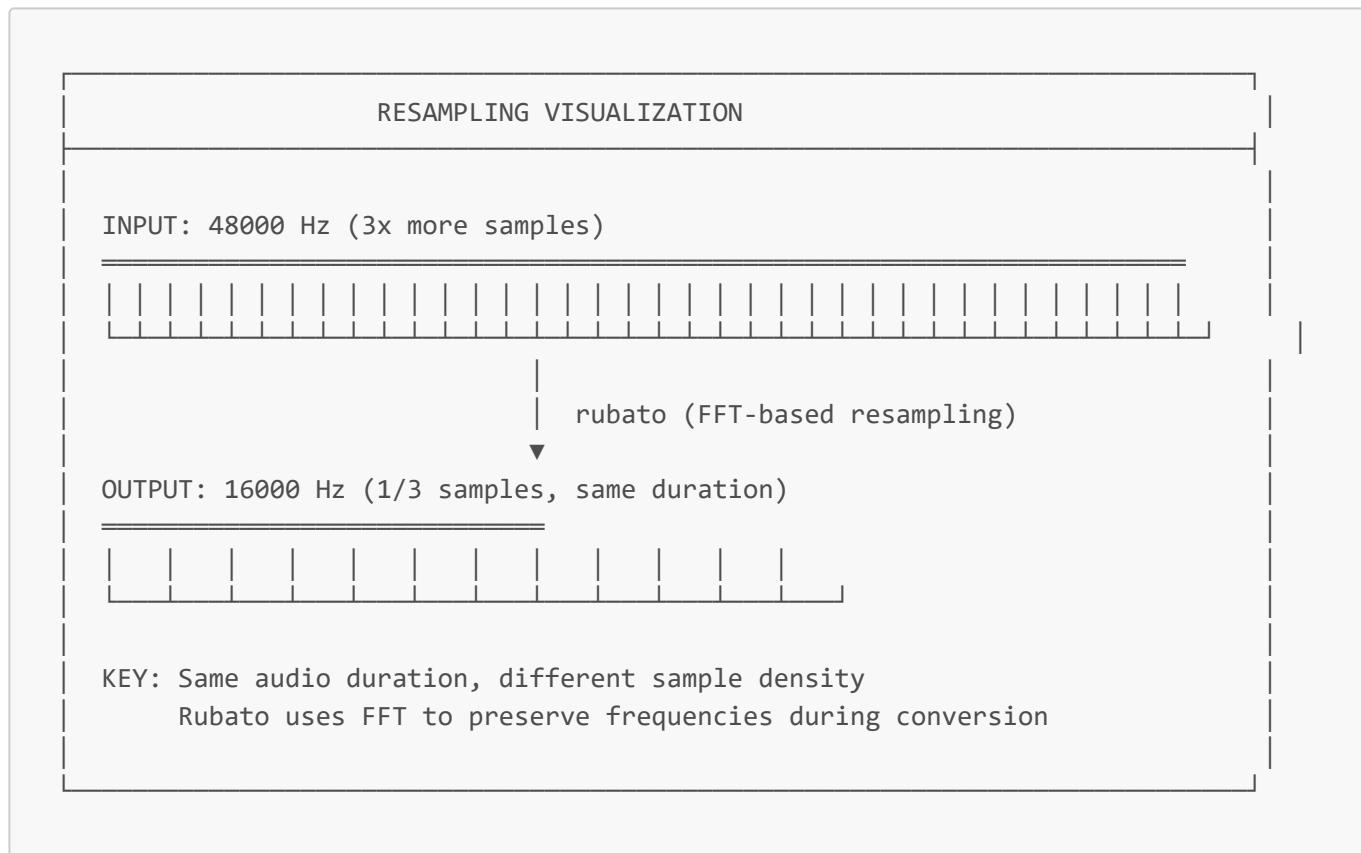
// Create resampler: 48000Hz → 16000Hz
let mut resampler = FftFixedIn::<f32>::new(
```

```

48000,    // Input sample rate
16000,    // Output sample rate (Whisper requires this)
1024,     // Chunk size
2,        // Sub-chunks (quality vs speed tradeoff)
1,        // Number of channels
)?;

// Resample audio
let input_frames = vec![audio_samples]; // Vec<Vec<f32>>
let output_frames = resampler.process(&input_frames, None)?;
let resampled: Vec<f32> = output_frames.into_iter().flatten().collect();

```

**Diagram:****AI/ML Crates****11. whisper-rs - Speech-to-Text (Whisper)**

```

# Base (CPU)
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }

# Windows/Linux x86_64 (GPU)
whisper-rs = { git = "...", features = ["cuda", "vulkan"] }

```

**What it is:** Rust bindings for OpenAI's Whisper speech recognition model via whisper.cpp (C++ implementation).

## Features:

- **cuda**: NVIDIA GPU acceleration
- **vulkan**: Cross-vendor GPU acceleration (NVIDIA, AMD, Intel)

## How it's used:

```
use whisper_rs::{
    FullParams, SamplingStrategy, WhisperContext, WhisperContextParameters,
};

// Load model (once at startup)
let ctx = WhisperContext::new_with_params(
    "/path/to/ggml-large-v3-turbo-q5_1.bin",
    WhisperContextParameters::default()
        .use_gpu(true), // Enable GPU if available
)?;

// Create transcription parameters
let mut params = FullParams::new(SamplingStrategy::Greedy { best_of: 1 });
params.set_language(Some("en"));
params.set_print_progress(false);
params.set_single_segment(true); // Real-time mode

// Create state and run inference
let mut state = ctx.create_state()?;
state.full(params, &audio_16khz)?; // Must be 16kHz mono f32!

// Extract result
let text = state.full_get_segment_text(0)?;
```

## 12. **parakeet-rs** - Speech-to-Text (Parakeet)

```
# Base
parakeet-rs = { version = "=0.3.0" }

# With GPU
parakeet-rs = { version = "=0.3.0", features = ["cuda", "directml"] }
```

**What it is:** Rust bindings for NVIDIA's Parakeet ASR models. Uses ONNX Runtime for inference.

## Model types:

- **Nemotron**: NVIDIA's Nemotron architecture
- **ParakeetTDT**: Token-and-Duration Transducer (fastest, most accurate)
- **Parakeet** (CTC): Connectionist Temporal Classification
- **ParakeetEOU**: End-of-Utterance detection

## How it's used:

```
use parakeet_rs::{ParakeetTDT, Transcriber, TimestampMode};

// Load model
let model = ParakeetTDT::from_files(
    "/path/to/encoder.onnx",
    "/path/to/decoder.onnx",
    "/path/to/joiner.onnx",
    "/path/to/tokenizer.model",
)?;

// Transcribe (expects 16kHz mono)
let text = model.transcribe(&audio_16khz, TimestampMode::None)?;
```

## 13. **ort** - ONNX Runtime

```
ort = { version = "2.0.0-rc.11", features = [
    "download-binaries", # Auto-download ONNX Runtime
    "cuda", # NVIDIA GPU
    "directml", # Windows GPU (AMD/Intel)
    "tensorrt", # NVIDIA optimized inference
    "xnnpack", # CPU optimization
] }
```

**What it is:** Rust bindings for Microsoft's ONNX Runtime - a high-performance inference engine for ONNX models.

**Why it's used:** Parakeet models are in ONNX format. ORT provides hardware acceleration across different GPUs.

### Feature breakdown:

Feature	What it does	Platform
cuda	NVIDIA GPU acceleration	Windows, Linux
directml	DirectX 12 GPU acceleration	Windows only
tensorrt	NVIDIA optimized inference	Windows, Linux
xnnpack	ARM/x86 CPU optimization	All
coreml	Apple Neural Engine	macOS only

## 14. **candle-*** - ML Framework (Gemma LLM)

```
candle-core = { git = "https://github.com/huggingface/candle.git", version =
"0.9.2" }
candle-nn = { git = "https://github.com/huggingface/candle.git", version = "0.9.2" }
candle-transformers = { git = "https://github.com/huggingface/candle.git", version
= "0.9.2" }
```

**What it is:** Hugging Face's pure-Rust ML framework. Like PyTorch, but in Rust.

#### Crate breakdown:

- `candle-core`: Tensor operations, device management, quantization
- `candle-nn`: Neural network layers
- `candle-transformers`: Pre-built transformer models (Gemma, Llama, etc.)

#### How it's used (Gemma LLM for grammar correction):

```
use candle_core::{Device, Tensor};
use candle_core::quantized::gguf_file;
use candle_transformers::models::quantized_gemma3 as model;
use candle_transformers::generation::LogitsProcessor;

// Load quantized GGUF model
let mut file = std::fs::File::open("/path/to/gemma-3-1b-it-q4_k_m.gguf")?;
let model_data = gguf_file::Content::read(&mut file)?;
let model = model::ModelWeights::from_gguf(model_data, &mut file, &Device::Cpu)?;

// Tokenize input
let tokens = tokenizer.encode(prompt, true)?;
let input_ids = Tensor::new(tokens.get_ids(), &Device::Cpu)?;

// Generate response
let logits = model.forward(&input_ids, 0)?;
let next_token = logits_processor.sample(&logits)?;
```

## 15. `tokenizers` - LLM Tokenization

```
tokenizers = "0.21.0"
```

**What it is:** Hugging Face's tokenizer library. Converts text ↔ token IDs for language models.

#### How it's used:

```
use tokenizers::Tokenizer;
```

```
// Load tokenizer from model config
let tokenizer = Tokenizer::from_file("/path/to/tokenizer.json")?;

// Encode text → tokens
let encoding = tokenizer.encode("Hello world", true)?;
let token_ids: Vec<u32> = encoding.get_ids().to_vec();
// [1, 15043, 2088] (example token IDs)

// Decode tokens → text
let text = tokenizer.decode(&token_ids, true)?;
// "Hello world"
```

## 16. **hf-hub** - Hugging Face Model Hub

```
hf-hub = "0.4"
```

**What it is:** Download models from Hugging Face Hub.

**How it's used:**

```
use hf_hub::api::sync::Api;

let api = Api::new()?;
let repo = api.model("google/gemma-3-1b-it-qat-q4_0-gguf".to_string());
let model_path = repo.get("gemma-3-1b-it-q4_0.gguf")?;
```

## Utility Crates

### 17. **chrono** - Date/Time

```
chrono = "0.4"
```

**What it is:** Date and time library for Rust.

**How it's used:**

```
use chrono::Local;

// Generate timestamp for file names
let timestamp = Local::now().format("%Y%m%d_%H%M%S");
let filename = format!("recording_{}.wav", timestamp);
// "recording_20260201_143052.wav"
```

## 18. **dirs** - OS Directories

```
dirs = "6.0.0"
```

**What it is:** Provides paths to standard OS directories.

**How it's used:**

```
use dirs;

// Get AppData directory (cross-platform)
let data_dir = dirs::data_dir();
// Windows: C:\Users\<user>\AppData\Roaming
// macOS:   /Users/<user>/Library/Application Support
// Linux:   /home/<user>/.local/share

// Get home directory
let home = dirs::home_dir();
// Windows: C:\Users\<user>
// macOS/Linux: /home/<user>
```

---

## 19. **rdev** - Global Hotkeys

```
rdev = "0.5"
```

**What it is:** Raw device events - captures keyboard/mouse events system-wide (even when app isn't focused).

**How it's used:**

```
use rdev::{listen, Event, EventType, Key};

// Listen for ALL keyboard events on the system
std::thread::spawn(|| {
    listen(|event| {
        match event.event_type {
            EventType::KeyPress(Key::ControlLeft) => {
                ctrl_held = true;
            }
            EventType::KeyPress(Key::MetaLeft) => {
                // Win key pressed
                if ctrl_held {
                    app_handle.emit("hotkey-start-recording", ());
                }
            }
        }
    })
})
```

```
        }
        EventType::KeyRelease(Key::ControlLeft) => {
            ctrl_held = false;
            if was_recording {
                app_handle.emit("hotkey-stop-recording", ());
            }
        }
    _ => {}
}).expect("Failed to listen for events");
});
```

**Security note:** Requires elevated permissions on some Linux systems.

---

## 20. **regex** - Regular Expressions

```
regex = "1.12.2"
```

**What it is:** Regular expression library for pattern matching in text.

**How it's used:**

```
use regex::Regex;

// Clean transcript text
let re = Regex::new(r"\s+").unwrap();
let cleaned = re.replace_all(transcript, " ");

// Remove [BLANK_AUDIO] artifacts
let re = Regex::new(r"\[BLANK_AUDIO\]").unwrap();
let cleaned = re.replace_all(&cleaned, "");
```

---

## 21. **anyhow** - Error Handling

```
anyhow = "1.0"
```

**What it is:** Easy error handling with automatic error context and conversion.

**How it's used:**

```
use anyhow::{Result, Context, bail};

fn load_model(path: &str) -> Result<Model> {
```

```

let file = std::fs::File::open(path)
    .context("Failed to open model file")?; // Adds context to error

if file.metadata()?.len() == 0 {
    bail!("Model file is empty"); // Create error and return early
}

Ok(Model::from_file(file)?)
}

```

## Why use anyhow?

- `?` works with any error type
  - `.context()` adds human-readable messages
  - `bail!()` macro for early returns
  - Better error messages in logs
- 

## Platform-Specific Dependencies

PLATFORM-SPECIFIC DEPENDENCY MATRIX				
	Windows x86_64	Windows ARM64	macOS (All)	Linux x86_64
whisper-rs	cuda, vulkan	(base)	(base)	cuda, vulkan
parakeet-rs	cuda, directml	directml	(base)	cuda
ort features	cuda, directml, tensorrt, xnnpack	directml, xnnpack	xnnpack	cuda, tensorrt, xnnpack

### LEGEND:

cuda = NVIDIA GPU (requires CUDA Toolkit)  
 vulkan = Cross-vendor GPU (requires Vulkan SDK)  
 directml = Windows GPU via DirectX 12 (built into Windows)  
 tensorrt = NVIDIA optimized inference (requires TensorRT)  
 xnnpack = CPU optimization (no extra requirements)

# 💻 OS Dependencies & Requirements

## Windows Requirements

### 1. Visual Studio Build Tools (Required)

Download: <https://visualstudio.microsoft.com/visual-cpp-build-tools/>

Required components:

- └─ MSVC v143 (or later) - C++ Build Tools
- └─ Windows 11 SDK (or Windows 10 SDK)
- └─ C++ CMake tools for Windows

**Why needed:** whisper-rs and parakeet-rs compile C++ code during build.

---

### 2. CUDA Toolkit (Optional - For NVIDIA GPU)

Download: <https://developer.nvidia.com/cuda-toolkit>

Required version: 12.x (tested with 12.9)

Sets environment variable:

CUDA_PATH = C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v12.9

**Why needed:** GPU acceleration for Whisper and Parakeet on NVIDIA cards.

**build.rs handles this:**

```
// Automatically finds CUDA libraries
if let Ok(cuda_path) = std::env::var("CUDA_PATH") {
    let lib_path = cuda_path.join("lib").join("x64");
    println!("cargo:rustc-link-search=native={}{}", lib_path.display());
}
```

### 3. Vulkan SDK (Optional - For AMD/Intel GPU with Whisper)

Download: <https://vulkan.lunarg.com/sdk/home>

Sets environment variable:

VULKAN_SDK = C:\VulkanSDK\1.3.xxx

**Why needed:** Alternative GPU acceleration for Whisper (works with AMD, Intel, NVIDIA).

#### 4. LLVM/Clang (Required for ARM64 Windows only)

```
Download: https://releases.llvm.org/
```

```
Required for: Windows ARM64 (Snapdragon laptops)
```

**Why needed:** whisper.cpp doesn't support MSVC on ARM64.

**build.rs handles this:**

```
#[cfg(all(target_os = "windows", target_arch = "aarch64"))]
{
    std::env::set_var("CC", "clang-cl");
    std::env::set_var("CXX", "clang-cl");
}
```

---

## macOS Requirements

### 1. Xcode Command Line Tools (Required)

```
xcode-select --install
```

**Why needed:** Provides C/C++ compiler (clang) and build tools.

---

### 2. Minimum macOS Version

```
Required: macOS 13.4+ (Ventura or later)
```

```
Why: ONNX Runtime requires macOS 13.4+ on Apple Silicon  
whisper.cpp requires C++17 std::filesystem (macOS 10.15+)
```

**build.rs handles this:**

```
#[cfg(target_os = "macos")]
{
    println!("cargo:rustc-env=MACOSX_DEPLOYMENT_TARGET=13.4");
    std::env::set_var("MACOSX_DEPLOYMENT_TARGET", "13.4");
}
```

## Linux Requirements

### 1. Build Essentials (Required)

```
# Debian/Ubuntu
sudo apt install build-essential cmake pkg-config

# Fedora
sudo dnf install @development-tools cmake

# Arch
sudo pacman -S base-devel cmake
```

### 2. ALSA Development Libraries (Required for audio)

```
# Debian/Ubuntu
sudo apt install libasound2-dev

# Fedora
sudo dnf install alsa-lib-devel

# Arch
sudo pacman -S alsa-lib
```

**Why needed:** cpal uses ALSA for audio capture on Linux.

### 3. X11 Development Libraries (Required for rdev hotkeys)

```
# Debian/Ubuntu
sudo apt install libx11-dev libxtst-dev

# Fedora
sudo dnf install libX11-devel libXtst-devel

# Arch
sudo pacman -S libx11 libxtst
```

**Why needed:** rdev uses X11 for capturing keyboard events.

### 4. CUDA Toolkit (Optional - For NVIDIA GPU)

```
# Follow NVIDIA's official guide:  
# https://developer.nvidia.com/cuda-downloads  
  
# Verify installation:  
nvcc --version  
nvidia-smi
```

---

## Dependency Installation Summary

### INSTALLATION CHECKLIST

#### WINDOWS:

- Visual Studio Build Tools (required)
- CUDA Toolkit 12.x (optional, for NVIDIA GPU)
- Vulkan SDK (optional, for Whisper GPU on AMD/Intel)
- LLVM/Clang (only for ARM64 Windows)

#### MACOS:

- Xcode Command Line Tools (required)
- macOS 13.4+ (required)

#### LINUX:

- build-essential, cmake (required)
- libasound2-dev (required for audio)
- libx11-dev, libxtst-dev (required for hotkeys)
- CUDA Toolkit (optional, for NVIDIA GPU)

#### ALL PLATFORMS:

- Rust (via rustup): <https://rustup.rs>
- Node.js 18+ (for frontend): <https://nodejs.org>
- Bun (faster npm alternative): <https://bun.sh>

---

## How Crates Map to Features

### FEATURE → CRATE MAPPING

#### FEATURE

#### CRATES INVOLVED

🎙️ Audio Capture

cpal, crossbeam-channel

Save Recordings	hound, dirs, chrono
Audio Preprocessing	rubato (resample to 16kHz)
Speech Recognition	whisper-rs OR parakeet-rs + ort
Grammar Correction	candle-*, tokenizers, hf-hub
Global Hotkeys	rdev
Persistent Settings	tauri-plugin-store, serde
System Tray	tauri (tray-icon feature)
Frontend ↔ Backend	tauri, serde, serde_json
Text Processing	regex
Error Handling	anyhow

## Import Patterns in Source Files

```
// _____
// COMMON IMPORT PATTERNS
// _____

// _____
// Tauri imports (commands, state, events)
// _____
use tauri::{AppHandle, State, Emitter, Manager};
use tauri::tray::TrayIconBuilder;

// _____
// Audio imports
// _____
use cpal::traits::{DeviceTrait, HostTrait, StreamTrait};
use crossbeam_channel::{unbounded, Sender, Receiver};
use hound::{WavSpec, WavWriter, WavReader, SampleFormat};
use rubato::{FftFixedIn, Resampler};

// _____
// AI/ML imports (Whisper)
// _____
use whisper_rs::{
    FullParams, SamplingStrategy, WhisperContext, WhisperContextParameters,
};

// _____
// AI/ML imports (Parakeet)
// _____
use parakeet_rs::{Nemotron, Parakeet, ParakeetEOU, ParakeetTDT, TimestampMode,
Transcriber};

// _____
// AI/ML imports (Candle for Gemma LLM)
// _____
use candle_core::{Device, Tensor};
use candle_core::quantized::gguf_file;
```

```
use candle_transformers::generation::LogitsProcessor;
use candle_transformers::models::quantized_gemma3 as model;
use tokenizers::Tokenizer;

// -----
// Utility imports
// -----
use std::sync::{Arc, Mutex};
use std::path::PathBuf;
use serde::{Serialize, Deserialize};
use chrono::Local;
use dirs;
use regex::Regex;
use rdev::{listen, Event, EventType, Key};
use anyhow::{Result, Context};

// -----
// Internal module imports
// -----
use crate::state::AudioState;
use crate::types::{ASREngine, AppState, TranscriptionChunk};
use crate::whisper::WhisperManager;
use crate::parakeet::ParakeetManager;
```

## ⚙️ Configuration Files Deep Dive

Complete breakdown of every configuration file and what each setting does for each platform.

### Configuration File Map

```
PROJECT CONFIGURATION FILES

ROOT/
├── package.json      # Frontend dependencies & npm scripts
├── tsconfig.json     # TypeScript compiler settings
└── vite.config.ts    # Vite dev server & bundler settings

└── src-tauri/
    ├── Cargo.toml      # Rust dependencies & platform configs
    ├── tauri.conf.json # Tauri app settings (window, bundle, build)
    ├── build.rs        # Build-time Rust code (CUDA paths, etc.)
    └── capabilities/
        └── default.json # Security permissions for frontend
```

## 📄 tauri.conf.json - Tauri Application Configuration

```
{  
  "$schema": "https://schema.tauri.app/config/2",  
  "productName": "taurscribe",  
  "version": "0.1.0",  
  "identifier": "abdul",  
  "build": { ... },  
  "app": { ... },  
  "bundle": { ... }  
}
```

### Line-by-Line Breakdown

#### Schema & Metadata (Lines 2-5)

```
"$schema": "https://schema.tauri.app/config/2",
```

**What it does:** Enables IDE autocomplete and validation for Tauri 2.x config format.

```
"productName": "taurscribe",
```

#### What it does:

- **Windows:** Creates `taurscribe.exe`, shown in Task Manager
- **macOS:** Creates `Taurscribe.app` bundle name
- **Linux:** Creates `taurscribe` binary name

```
"version": "0.1.0",
```

**What it does:** Application version shown in:

- **Windows:** File Properties → Details → Product Version
- **macOS:** About dialog, App Store
- **Linux:** Package metadata

```
"identifier": "abdul",
```

**What it does:** Unique app identifier (should be reverse domain like `com.yourname.tauri`).

- **Windows:** Used for registry keys, app data paths
- **macOS:** Bundle identifier (required for App Store)

- **Linux:** Desktop entry identifier

## Build Configuration (Lines 6-11)

```
"build": {
  "beforeDevCommand": "bun run dev",
  "devUrl": "http://localhost:1420",
  "beforeBuildCommand": "bun run build",
  "frontendDist": "../dist"
}
```

Setting	What it does	When it runs
beforeDevCommand	Runs <code>bun run dev</code> → starts Vite dev server	<code>bun tauri dev</code>
devUrl	Tells Tauri to load frontend from this URL	Development only
beforeBuildCommand	Runs <code>bun run build</code> → creates production bundle	<code>bun tauri build</code>
frontendDist	Where to find built frontend files	Production build

## Development Flow:

- ```
bun tauri dev
```
1. beforeDevCommand: "bun run dev"
    - └ Starts Vite at `http://localhost:1420`
  2. Tauri compiles Rust backend
    - └ `cargo build --features ...`
  3. Opens window pointing to devUrl
    - └ WebView loads `http://localhost:1420`
  4. Hot Module Replacement (HMR) active
    - └ Edit React → instant update (no restart)

## Production Build Flow:

- ```
bun tauri build
```
1. beforeBuildCommand: "bun run build"

- ```

    └── Vite creates optimized bundle in ../dist

2. Tauri compiles Rust in release mode
   └── cargo build --release --features ...

3. Bundles frontend from frontendDist
   └── Embeds ../dist/* into the binary

4. Creates platform installers
   └── Windows: .msi, .exe
   └── macOS:   .app, .dmg
   └── Linux:   .deb, .AppImage

```

## App Configuration (Lines 12-23)

```

"app": {
  "windows": [
    {
      "title": "Taurscribe",
      "width": 600,
      "height": 600
    }
  ],
  "security": {
    "csp": null
  }
}

```

| Setting   | What it does                                            |
|-----------|---------------------------------------------------------|
| windows   | Array of window configurations                          |
| title     | Window title bar text                                   |
| width     | Initial window width in pixels                          |
| height    | Initial window height in pixels                         |
| csp: null | Disables Content Security Policy (allows all resources) |

### Additional Window Options (not used but available):

```
{
  "title": "Taurscribe",
  "width": 600,
  "height": 600,
  "resizable": true,           // Allow resize
}
```

```

"fullscreen": false,           // Start fullscreen
"center": true,              // Center on screen
"decorations": true,          // Show title bar
"transparent": false,         // Transparent background
"alwaysOnTop": false,         // Stay on top
"minWidth": 400,              // Minimum width
"minHeight": 300              // Minimum height
}

```

## CSP (Content Security Policy):

```

"csp": null // Current: Allow everything (development convenience)

// Production recommendation:
"csp": "default-src 'self'; script-src 'self'"

```

Setting `csp: null` is **unsafe for production** but convenient for development.

## Bundle Configuration (Lines 24-34)

```

"bundle": {
  "active": true,
  "targets": "all",
  "icon": [
    "icons/32x32.png",
    "icons/128x128.png",
    "icons/128x128@2x.png",
    "icons/icon.icns",
    "icons/icon.ico"
  ]
}

```

| Setting              | What it does                            |
|----------------------|-----------------------------------------|
| <code>active</code>  | Enable bundling (create installers)     |
| <code>targets</code> | Which installer formats to create       |
| <code>icon</code>    | App icons for different platforms/sizes |

## Icon Usage by Platform:

| File                   | Used On | Purpose                  |
|------------------------|---------|--------------------------|
| <code>icon.ico</code>  | Windows | Executable icon, taskbar |
| <code>icon.icns</code> | macOS   | App bundle icon, Dock    |

| File           | Used On | Purpose             |
|----------------|---------|---------------------|
| 32x32.png      | Linux   | Small icon (menus)  |
| 128x128.png    | Linux   | Application icon    |
| 128x128@2x.png | macOS   | Retina display icon |

### Bundle Targets by Platform:

```
BUNDLE OUTPUTS (targets: "all")

WINDOWS:
└── target/release/bundle/
    ├── msi/taurscribe_0.1.0_x64_en-US.msi      # MSI installer
    └── nsis/taurscribe_0.1.0_x64-setup.exe       # NSIS installer

MACOS:
└── target/release/bundle/
    ├── macos/Taurscribe.app                  # App bundle
    └── dmg/Taurscribe_0.1.0_x64.dmg          # Disk image

LINUX:
└── target/release/bundle/
    ├── deb/tauruscribe_0.1.0_amd64.deb        # Debian package
    ├── rpm/tauruscribe-0.1.0.x86_64.rpm        # RPM package
    └── appimage/tauruscribe_0.1.0_amd64.AppImage # Portable
```

## 📦 Cargo.toml - Rust Dependencies & Platform Configuration

### File Structure Overview

```
[package]           # Package metadata
[lib]              # Library output settings
[build-dependencies] # Compile-time dependencies
[dependencies]     # Runtime dependencies (ALL platforms)
[target.'cfg(...)' .dependencies] # Platform-specific deps
[profile.dev]       # Development build settings
```

### Package Section (Lines 1-6)

```
[package]
name = "taurscribe"
version = "0.1.0"
description = "A Tauri App"
authors = ["you"]
edition = "2021"
```

| Field       | What it does                                          |
|-------------|-------------------------------------------------------|
| name        | Crate name (used in <code>use taurcribe::...</code> ) |
| version     | Crate version (for dependency resolution)             |
| description | Shows in <code>cargo search</code> , crates.io        |
| authors     | Package authors                                       |
| edition     | Rust edition (2021 = latest stable features)          |

### Rust Editions:

- 2015: Original Rust
  - 2018: Module system improvements, `async/await`
  - 2021: Disjoint closure captures, `IntoIterator` for arrays
- 

### Library Section (Lines 10-15)

```
[lib]
name = "taurscribe_lib"
crate-type = ["staticlib", "cdylib", "rlib"]
```

| Setting    | What it does                                                           |
|------------|------------------------------------------------------------------------|
| name       | Output library name (different from package to avoid Windows conflict) |
| crate-type | What kinds of libraries to produce                                     |

### Crate Types Explained:

| Type      | Output                                                                                       | Used For              |
|-----------|----------------------------------------------------------------------------------------------|-----------------------|
| rlib      | <code>libtaurscribe_lib.rlib</code>                                                          | Rust-to-Rust linking  |
| cdylib    | <code>taurscribe_lib.dll</code> (Win) / <code>.so</code> (Linux) / <code>.dylib</code> (Mac) | C/FFI dynamic linking |
| staticlib | <code>taurscribe_lib.lib</code> (Win) / <code>.a</code> (Unix)                               | C/FFI static linking  |

**Why all three?** Tauri needs different link types for different build scenarios.

---

## Dependencies Section (Lines 20-46)

These dependencies are used on **ALL platforms**:

```
[dependencies]
# =====
# FRAMEWORK
# =====
tauri = { version = "2", features = ["tray-icon", "image-png"] }
tauri-plugin-opener = "2"      # Open URLs/files with default app
tauri-plugin-fs = "2"          # File system access
tauri-plugin-store = "2.4.2"   # Persistent JSON settings

# =====
# SERIALIZATION
# =====
serde = { version = "1", features = ["derive"] }  # Derive Serialize/Deserialize
serde_json = "1"                                # JSON parsing

# =====
# AUDIO
# =====
cpal = "0.15"          # Cross-platform audio capture
crossbeam-channel = "0.5" # Thread communication
hound = "3.5"           # WAV file I/O
rubato = "0.14"         # Audio resampling

# =====
# UTILITIES
# =====
chrono = "0.4"          # Date/time
dirs = "6.0.0"            # OS directories
rdev = "0.5"              # Global hotkeys
regex = "1.12.2"          # Text patterns
anyhow = "1.0"             # Error handling

# =====
# LLM (GRAMMAR CORRECTION)
# =====
candle-core = { git = "...", version = "0.9.2" }
candle-nn = { git = "...", version = "0.9.2" }
candle-transformers = { git = "...", version = "0.9.2" }
tokenizers = "0.21.0"
hf-hub = "0.4"
```

---

## Platform-Specific Dependencies

This is where Cargo.toml gets interesting - different platforms get different dependencies!

### How `cfg()` Works

```
[target.'cfg(CONDITION)'.dependencies]
```

| Condition               | Matches          |
|-------------------------|------------------|
| target_os = "windows"   | Any Windows      |
| target_os = "macos"     | Any macOS        |
| target_os = "linux"     | Any Linux        |
| target_arch = "x86_64"  | 64-bit Intel/AMD |
| target_arch = "aarch64" | 64-bit ARM       |
| all(A, B)               | Both A AND B     |
| any(A, B)               | Either A OR B    |

### Base Dependencies (Lines 51-55) - Fallback for All Platforms

```
# 1. Base / CPU Fallback
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
parakeet-rs = { version = "=0.3.0" }
ort = { version = "2.0.0-rc.11", features = ["download-binaries"] }
```

**No GPU features** - pure CPU. Used when no platform-specific override matches.

### macOS (Lines 66-68) - Apple Silicon & Intel

```
[target.'cfg(target_os = "macos")'.dependencies]
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
ort = { version = "2.0.0-rc.11", features = ["download-binaries", "xnnpack"] }
```

#### Feature    What it does

xnnpack    Optimized CPU inference (works on both Intel and Apple Silicon)

**Note:** No **cuda** or **vulkan** - Macs don't have NVIDIA GPUs. Apple Silicon uses Metal via CoreML internally.



App → ONNX Runtime → CoreML Provider → Apple Neural Engine (ANE)

Whisper: CPU (whisper.cpp uses Accelerate framework)

Parakeet: CoreML via ONNX Runtime

INTEL MAC:

App → ONNX Runtime → XNNPACK Provider → CPU (optimized SIMD)

Whisper: CPU (whisper.cpp)

Parakeet: XNNPACK (CPU optimized)

## Windows x86\_64 (Lines 77-91) - Full GPU Support

```
[target.'cfg(all(target_os = "windows", target_arch = "x86_64"))'.dependencies]

whisper-rs = { git = "...", features = ["cuda", "vulkan"] }
parakeet-rs = { version = "=0.3.0", features = ["cuda", "directml"] }
ort = { version = "2.0.0-rc.11", features = [
    "download-binaries",
    "cuda",
    "directml",
    "tensorrt",
    "xnnpack",
] }
```

| Feature  | GPU Vendor         | What it does                    |
|----------|--------------------|---------------------------------|
| cuda     | NVIDIA             | Direct GPU compute via CUDA     |
| vulkan   | NVIDIA, AMD, Intel | Cross-vendor GPU via Vulkan API |
| directml | AMD, Intel, NVIDIA | DirectX 12 GPU acceleration     |
| tensorrt | NVIDIA             | Optimized inference engine      |
| xnnpack  | CPU                | Fallback if no GPU              |

WINDOWS x86\_64 ACCELERATION STACK

NVIDIA GPU (RTX/GTX):

Whisper: whisper.cpp → CUDA → NVIDIA GPU

Parakeet: ONNX Runtime → CUDA Provider → NVIDIA GPU  
OR: ONNX Runtime → TensorRT Provider → NVIDIA GPU

AMD GPU (RX/Radeon):

Whisper: whisper.cpp → Vulkan → AMD GPU  
Parakeet: ONNX Runtime → DirectML Provider → AMD GPU

INTEL GPU (Arc/Iris):

Whisper: whisper.cpp → Vulkan → Intel GPU  
Parakeet: ONNX Runtime → DirectML Provider → Intel GPU

NO GPU / Fallback:

Whisper: whisper.cpp → CPU  
Parakeet: ONNX Runtime → XNNPACK Provider → CPU

## Linux x86\_64 (Lines 98-111) - NVIDIA Focus

```
[target.'cfg(all(target_os = "linux", target_arch = "x86_64"))'.dependencies]

whisper-rs = { git = "...", features = ["cuda", "vulkan"] }
parakeet-rs = { version = "=0.3.0", features = ["cuda"] }
ort = { version = "2.0.0-rc.11", features = [
    "download-binaries",
    "cuda",
    "tensorrt",
    "xnnpack",
] }
```

**Note:** No `directml` - that's Windows-only (DirectX).

### LINUX x86\_64 ACCELERATION STACK

NVIDIA GPU:

Whisper: whisper.cpp → CUDA → NVIDIA GPU  
Parakeet: ONNX Runtime → CUDA/TensorRT Provider → NVIDIA GPU

AMD/INTEL GPU:

```
Whisper: whisper.cpp → Vulkan → AMD/Intel GPU
Parakeet: ONNX Runtime → XNNPACK (CPU fallback, no DirectML on Linux)
```

## Windows ARM64 (Lines 116-122) - Snapdragon/Qualcomm

```
[target.'cfg(all(target_os = "windows", target_arch = "aarch64"))'.dependencies]
ort = { version = "2.0.0-rc.11", features = [
    "download-binaries",
    "directml",
    "xnnpack",
] }
parakeet-rs = { version = "=0.3.0", features = ["directml"] }
```

**Note:** No **cuda** or **vulkan** - ARM Windows devices (Surface Pro X, Snapdragon laptops) don't have NVIDIA GPUs.

### WINDOWS ARM64 ACCELERATION STACK

QUALCOMM SNAPDRAGON (Adreno GPU):

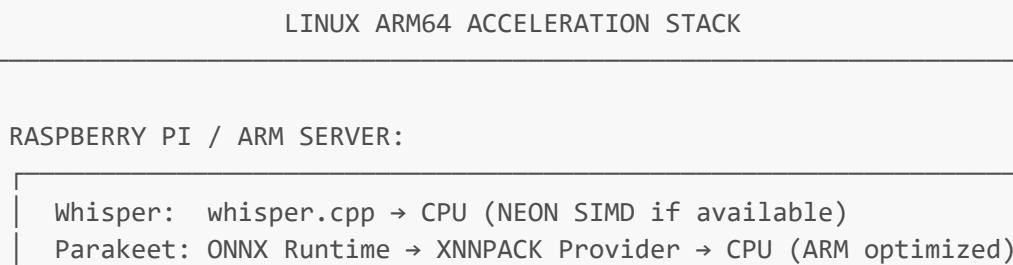
```
Whisper: whisper.cpp → CPU (no ARM CUDA)
Parakeet: ONNX Runtime → DirectML Provider → Adreno GPU
```

DirectML automatically uses the integrated GPU on ARM Windows devices

## Linux ARM64 (Lines 127-130) - Raspberry Pi, etc.

```
[target.'cfg(all(target_os = "linux", target_arch = "aarch64"))'.dependencies]
whisper-rs = { git = "https://codeberg.org/tazz4843/whisper-rs.git" }
parakeet-rs = { version = "=0.3.0" }
ort = { version = "2.0.0-rc.11", features = ["download-binaries", "xnnpack"] }
```

**CPU only** - no GPU acceleration. XNNPACK provides ARM-optimized CPU inference.



## Profile Section (Lines 133-141)

```

[profile.dev]
opt-level = 0          # Don't optimize YOUR code (fast compile)

[profile.dev.package.*]
opt-level = 1          # Lightly optimize dependencies (faster runtime)

```

### What this does:

| Profile            | Your Code     | Dependencies  | Result                     |
|--------------------|---------------|---------------|----------------------------|
| Default dev        | opt-level = 0 | opt-level = 0 | Fast compile, slow runtime |
| <b>This config</b> | opt-level = 0 | opt-level = 1 | Fast compile, OK runtime   |
| Release            | opt-level = 3 | opt-level = 3 | Slow compile, fast runtime |

### Optimization Levels:

- **0**: No optimization (fastest compile)
- **1**: Basic optimization (small speedup)
- **2**: Standard optimization
- **3**: Aggressive optimization (slowest compile)
- **"s"**: Optimize for size
- **"z"**: Optimize for size more aggressively

## 🔧 build.rs - Build-Time Configuration

This file runs **before** compilation and configures platform-specific settings.

```
fn main() {
    tauri_build::build(); // Standard Tauri build

    // Platform-specific configurations below...
}
```

## macOS Configuration

```
#[cfg(target_os = "macos")]
{
    println!("cargo:rustc-env=MACOSX_DEPLOYMENT_TARGET=13.4");
    std::env::set_var("MACOSX_DEPLOYMENT_TARGET", "13.4");
    std::env::set_var("CMAKE OSX DEPLOYMENT TARGET", "13.4");
}
```

### What it does:

- Sets minimum macOS version to 13.4 (Ventura)
  - Required because ONNX Runtime needs macOS 13.4+ on Apple Silicon
  - `CMAKE OSX DEPLOYMENT TARGET` ensures CMake-based deps (whisper-rs) also target 13.4
- 

## Windows ARM64 Configuration

```
#[cfg(all(target_os = "windows", target_arch = "aarch64"))]
{
    std::env::set_var("CC", "clang-cl");
    std::env::set_var("CXX", "clang-cl");
    std::env::set_var("CMAKE_GENERATOR_TOOLSET", "ClangCL");
}
```

### What it does:

- Forces Clang compiler instead of MSVC
  - Required because whisper.cpp doesn't compile with MSVC on ARM64
  - User must install LLVM/Clang for Windows ARM64
- 

## Windows CUDA Configuration

```
#[cfg(windows)]
{
    // Try CUDA_PATH environment variable
    if let Ok(cuda_path) = std::env::var("CUDA_PATH") {
        let lib_path = cuda_path.join("lib").join("x64");
        if lib_path.exists() {
```

```

        println!("cargo:rustc-link-search=native={}", lib_path.display());
    }

    // Fallback: Check standard installation path
    let fallback = r"C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v12.9\lib\x64";
    // ...
}

```

**What it does:**

- Tells the Rust linker where to find CUDA libraries (cUBLAS.lib, etc.)
- First checks `CUDA_PATH` environment variable (set by CUDA installer)
- Falls back to standard installation path if not found
- Without this, you get `LNK1181: cannot open input file 'cUBLAS.lib'`

 capabilities/default.json - Security Permissions

```
{
  "$schema": "../gen/schemas/desktop-schema.json",
  "identifier": "default",
  "description": "Capability for the main window",
  "windows": ["main"],
  "permissions": [
    "core:default",
    "opener:default",
    "store:default"
  ]
}
```

**What it does:** Defines what the frontend JavaScript can do.

| Permission                  | What it allows                                             |
|-----------------------------|------------------------------------------------------------|
| <code>core:default</code>   | Basic Tauri operations (invoke commands, listen to events) |
| <code>opener:default</code> | Open URLs in browser, files with default app               |
| <code>store:default</code>  | Read/write to persistent store (settings.json)             |

**Without these permissions:**

```

// This would FAIL without core:default
await invoke("start_recording");

// This would FAIL without store:default
const store = await Store.load("settings.json");

```

## 📦 package.json - Frontend Configuration

```
{
  "name": "taurscribe",
  "private": true,
  "version": "0.1.0",
  "type": "module",
  "scripts": { ... },
  "dependencies": { ... },
  "devDependencies": { ... }
}
```

### Scripts Section

```
"scripts": {
  "dev": "vite",
  "build": "tsc && vite build",
  "preview": "vite preview",
  "tauri": "tauri",
  "check": "cd src-tauri && cargo check"
}
```

| Script  | Command         | What it does                            |
|---------|-----------------|-----------------------------------------|
| dev     | bun run dev     | Start Vite dev server (port 1420)       |
| build   | bun run build   | Type-check then build production bundle |
| preview | bun run preview | Preview production build locally        |
| tauri   | bun tauri ...   | Run Tauri CLI commands                  |
| check   | bun run check   | Type-check Rust code only               |

### Dependencies Section

```
"dependencies": {
  "@tauri-apps/api": "^2",           // Core Tauri API (invoke, listen)
  "@tauri-apps/plugin-opener": "^2",  // Open URLs/files
  "@tauri-apps/plugin-store": "^2.4.2", // Persistent settings
  "react": "^19.1.0",                // React UI library
  "react-dom": "^19.1.0",            // React DOM renderer
  "sonner": "^2.0.7"                 // Toast notifications
}
```

## DevDependencies Section

```
"devDependencies": {
  "@types/react": "^19.1.8",           // React TypeScript types
  "@types/react-dom": "^19.1.6",        // ReactDOM TypeScript types
  "@vitejs/plugin-react": "^4.6.0",    // Vite React plugin (JSX, HMR)
  "typescript": "~5.8.3",              // TypeScript compiler
  "vite": "^7.0.4",                   // Build tool & dev server
  "@tauri-apps/cli": "^2"             // Tauri command-line tools
}
```

## ⚡ vite.config.ts - Vite Configuration

```
export default defineConfig(async () => ({
  plugins: [react()],           // Enable React JSX/TSX support
  clearScreen: false,            // Don't clear terminal (show Rust errors)
  server: {
    port: 1420,                 // Fixed port (Tauri expects this)
    strictPort: true,            // Fail if port busy (don't auto-pick another)
    host: host || false,         // Bind to TAURI_DEV_HOST if set
    hmr: host ? {                // Hot Module Replacement config
      protocol: "ws",
      host,
      port: 1421,
    } : undefined,
    watch: {
      ignored: ["**/src-tauri/**"], // Don't watch Rust files
    },
  },
}));
```

| Setting                         | Why it matters for Tauri                                       |
|---------------------------------|----------------------------------------------------------------|
| port: 1420                      | Must match <code>devUrl</code> in <code>tauri.conf.json</code> |
| strictPort: true                | Prevents Vite from using 1421 if 1420 is busy                  |
| clearScreen: false              | Shows Rust compile errors in terminal                          |
| ignored: <code>src-tauri</code> | Prevents Vite restart when Rust files change                   |

## tsconfig.json - TypeScript Configuration

```
{
  "compilerOptions": {
    "target": "ES2020",           // Output JS version
    "lib": ["ES2020", "DOM"],     // Available APIs
  }
}
```

```

"module": "ESNext",           // Module system
"moduleResolution": "bundler", // How to find modules
"jsx": "react-jsx",          // JSX transform (React 17+)
"strict": true,               // Enable all strict checks
"noUnusedLocals": true,       // Error on unused variables
"noEmit": true                // Don't emit (Vite handles this)
},
"include": ["src"]           // Only check src/ folder
}

```

## Platform Support Summary

PLATFORM SUPPORT MATRIX

| Platform | Architecture         | GPU Support                            | Output Format         |
|----------|----------------------|----------------------------------------|-----------------------|
| Windows  | x86_64               | CUDA<br>Vulkan<br>DirectML<br>TensorRT | .msi, .exe (NSIS)     |
| Windows  | ARM64                | DirectML<br>XNNPACK                    | .msi, .exe (NSIS)     |
| macOS    | x86_64<br>(Intel)    | XNNPACK                                | .app, .dmg            |
| macOS    | ARM64<br>(Apple Si)  | CoreML<br>XNNPACK                      | .app, .dmg            |
| Linux    | x86_64               | CUDA<br>Vulkan<br>TensorRT             | .deb, .rpm, .AppImage |
| Linux    | ARM64<br>(RPi, etc.) | XNNPACK                                | .deb, .rpm, .AppImage |

GPU ACCELERATION PRIORITY (runtime selection):

1. CUDA (NVIDIA)
2. TensorRT (NVIDIA optimized)
3. DirectML (Windows any GPU)
4. Vulkan (Cross-platform GPU)
5. CoreML (Apple Silicon)
6. XNNPACK (CPU optimized)
7. CPU (fallback)



---

Happy coding! 🐍