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

What is Taurscribe?

🎬 Movie Theater Analogy

Imagine Taurscribe is like a **movie theater with live subtitles**:

```
┌─────────────────────────────────────────────────────────────┐
│                    🎬 TAURSCRIBE THEATER                      │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  🎤 ACTOR (You speaking)                                     │
│       │                                                     │
│       │ Your voice travels through the air                 │
│       ▼                                                     │
│  🎧 SOUND ENGINEER (Microphone + Audio Processing)           │
│       │                                                     │
│       │ Captures and prepares the sound                    │
│       ▼                                                     │
│  ⚡ TRANSCRIBER #1 (Parakeet - Speed)                        │
│       │   "I write instantly but might miss details"       │
│       │                                                     │
│  🎯 TRANSCRIBER #2 (Whisper - Accuracy)                      │
│      │   "I wait 6 seconds but write perfectly"           │
│      ▼                                                     │
│  📝 EDITOR (LLM Grammar Correction)                        │
│      │   "I fix any grammar mistakes"                      │
│      ▼                                                     │
│  🔤 PROOFREADER (Spell Checker)                            │
│      │   "I catch any spelling errors"                     │
│      ▼                                                     │
│  📺 SUBTITLE SCREEN (Frontend UI)                          │
│         "The audience sees the final text!"               │
│                                                            │
└────────────────────────────────────────────────────────────┘
```

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

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
- **Post-Processing**:
  - ✨ **LLM** - Grammar & style correction with fine-tuned Qwen 2.5 0.5B (GGUF)
  - 🔤 **Spell Check** - Catch any spelling mistakes with SymSpell

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)
- ✅ Grammar & style correction with local fine-tuned LLM (CPU or GPU)
- ✅ Spell checking for final polish
- ✅ Model download manager (download models from within the app)
- ✅ Global hotkey (Ctrl+Win) works from any application

The Big Picture

🏭 Factory Analogy

Think of Taurscribe as a **speech-to-text factory**:

```
═══════════════════════════════════════════════════════════════════════
                    🏭 TAURSCRIBE FACTORY OVERVIEW
```

```
═════════════════════════════════════════════════════════════════════

  RAW MATERIAL                    PROCESSING                     OUTPUT
  ════════════                    ══════════                     ══════

  🎤 Your Voice             ┌─────────────────────────┐
      │                    │    FRONTEND (React)     │           📺 UI
      │                    │    App.tsx + hooks/     │           Display
      │                    │    =====================│              ▲
      │                    │    • Recording buttons  │              │
      │                    │    • Model selection    │              │
      │                    │    • Settings modal     │              │
      │                    │    • Transcription view │              │
      │                    │    (logic split into 5  │              │
      │                    │     custom hooks)       │              │
      │                    └────────────┬────────────┘              │
     │                                 │                           │
     │                    Tauri IPC Bridge (JavaScript ↔ Rust)     │
     │                                 │                           │
     ▼                    ┌────────────▼────────────┐              │
  ════════════            │     BACKEND (Rust)      │              │
  │ Microphone │─────────►│     lib.rs - 132 lines  │──────────────┘
  ════════════            │     ====================│
                          │     Entry point, setup  │
                          └────────────┬────────────┘
                                       │
              ┌────────────────────────┼────────────────────────┐
              │                        │                        │
              ▼                        ▼                        ▼
     ┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
     │  whisper.rs     │    │  parakeet.rs    │    │  vad.rs         │
     │  (Whisper AI)   │    │  (Parakeet AI)  │    │  (Silence Det.) │
     │  ~630 lines     │    │  ~339 lines     │    │  ~162 lines     │
     └─────────────────┘    └─────────────────┘    └─────────────────┘
               │                        │
               │              ┌─────────────────┐
               │              │parakeet_loaders │
               │              │(GPU/CPU loaders)│
               │              │  ~300 lines     │
               │              └─────────────────┘
              │                        │                        │
              └────────────────────────┼────────────────────────┘
                                       │
                                       ▼
                          ┌─────────────────────────┐
                          │    POST-PROCESSING      │
                          │    llm.rs (Grammar)     │
                          │    spellcheck.rs        │
                          └─────────────────────────┘

═════════════════════════════════════════════════════════════════════
```

🔄 Simple Data Flow

```
🎤 Your Voice
    │
    ├──► Microphone captures sound waves (48kHz stereo)
    │
    ├──► Converts to numbers (audio samples: -1.0 to 1.0)
    │
    ├──► Resamples to 16kHz mono (AI requirement)
    │
    ├──► Split into two streams:
    │
    ├──► Stream 1 → 💾 Save to disk (WAV file)
    │
    └──► Stream 2 → 🤖 AI transcription → 📝 Text
                          │
                          ▼
                    ✨ Grammar Correction (LLM)
                          │
                          ▼
                    🔤 Spell Check
                          │
                          ▼
                    📺 Display to User
```

⚠️ Gotcha: Why Two Audio Streams?

Common Mistake: Beginners often ask "Why not just use one stream?"

Answer: The WAV file is saved in **original quality** (48kHz stereo) while the AI needs **processed audio** (16kHz mono). If we only kept the processed version, we'd lose quality. By saving the original, you can:

- Re-transcribe with different settings later
- Share the original recording
- Use it for other purposes

🖥️ Platform Support & Hardware Acceleration

🚙 Car Engine Analogy

Think of hardware acceleration like **different car engines**:

```
┌─────────────────────────────────────────────────────────────────────┐
│                   🚙 ACCELERATION COMPARISON                          │
├─────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  ⚡ CUDA (NVIDIA GPU)     = Tesla Electric (0-60 in 2s)             │
│     Fastest when available, requires NVIDIA                         │
│                                                                      │
│  🌋 Vulkan (Any GPU)      = Sports Car (0-60 in 4s)            │
│     Good speed, works with AMD/Intel too                       │
│                                                                 │
│  🪟 DirectML (Windows)    = Modern Sedan (0-60 in 5s)          │
│     Windows universal, works with NPUs                         │
│                                                                 │
│  🍎 CoreML (Apple)        = BMW Electric (0-60 in 3s)          │
│     Mac-optimized, uses Neural Engine                          │
│                                                                 │
│  💨 XNNPACK (CPU)         = Economy Car (0-60 in 8s)           │
│     Works everywhere, uses SIMD                                 │
│                                                                 │
│  🐢 Pure CPU              = Bicycle (0-60 in... eventually)    │
│     Always available as fallback                                │
│                                                                 │
└────────────────────────────────────────────────────────────────┘
```

📊 Platform Matrix

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

🔍 How GPU Detection Works

```
┌─────────────────────────────────────────────────────────────┐
│                 GPU DETECTION FLOW (whisper.rs)              │
└─────────────────────────────────────────────────────────────┘
                              │
                              ▼
                    ┌─────────────────┐
                    │  Try GPU Mode   │
                    │  use_gpu(true)  │
                    └─────────────────┘
                             │
                             │
                    ┌───────▼───────┐
                    │  Load Model   │
                    │  with GPU     │
                    └───────┬───────┘
                            │
              ┌─────────────┴─────────────┐
              │                           │
           Success?                    Failure?
              │                           │
              ▼                           ▼
      ┌───────────────┐           ┌───────────────┐
      │ Run nvidia-smi│           │ Try CPU Mode  │
      │ command       │           │ use_gpu(false)│
      └───────┬───────┘           └───────┬───────┘
              │                           │
      ┌───────▼───────┐                   ▼
      │Command exists?│            Return CPU Backend
      └───────┬───────┘
              │
      ┌───────┴───────┐
      │               │
     Yes              No
      │               │
      ▼               ▼
    CUDA           Vulkan
```

⚠ Gotcha: CUDA Requires nvidia-smi

Common Mistake: "I have an NVIDIA GPU but it's using Vulkan!"

Solution: Make sure NVIDIA drivers are properly installed. The detection runs:

```
std::process::Command::new("nvidia-smi").output()
```

If this fails, Taurscribe assumes Vulkan is available instead.

🎙 Audio Processing: Whisper vs Parakeet

🍕 Pizza Delivery Analogy

```
┌─────────────────────────────────────────────────────────┐
│                 🍕 AUDIO PROCESSING COMPARISON            │
├─────────────────────────────────────────────────────────┤
│                                                          │
│  🧠 WHISPER AI = Traditional Pizza Delivery             │
│                                                          │
│     • Waits for full order (6 seconds of audio)         │
│     • Checks if pizza is worth delivering (VAD check)   │
│     • Delivers high-quality pizza (accurate transcription)          │
│     • Latency: 6+ seconds                                           │
│                                                                     │
│  ⚡ PARAKEET = Speed Delivery Service                               │
│                                                                     │
│     • Delivers slices as they're ready (0.56s chunks)               │
│     • No quality check (skips VAD for speed)                        │
│     • Words appear almost instantly                                 │
│     • Latency: ~0.6 seconds                                         │
│                                                                     │
└─────────────────────────────────────────────────────────────────────┘
```

📊 Technical Comparison

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	✅ Yes (energy-based)	❌ No (speed priority)
Context	Manual (we provide previous text)	Automatic (built-in state via `m.reset()`)
GPU Support	CUDA, Vulkan, CPU	CUDA, DirectML, CPU
Model Format	GGML (.bin files)	ONNX (.onnx files)
Accuracy	Excellent (95-98%)	Very Good (92-96%)
Best For	Meetings, lectures	Live streaming, gaming

🔄 Whisper Processing Pipeline

```
═══════════════════════════════════════════════════════════════════════
                        🎤 WHISPER PIPELINE
═══════════════════════════════════════════════════════════════════════

STEP 1: 🎤 MICROPHONE CAPTURE
┌─────────────────────────────────────────────────────────────────────┐
│ Raw Audio: 48,000 samples/second, Stereo, Float32                   │
│ Example: [0.01, -0.02, 0.03, -0.01, 0.04, ...]                      │
└─────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 2: 🎛 CONVERT TO MONO
┌─────────────────────────────────────────────────────────────────────┐
│ Stereo [L1, R1, L2, R2] → Mono [(L1+R1)/2, (L2+R2)/2]               │
│ Why? AI models expect single-channel audio                          │
└─────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 3: 🔄 RESAMPLE (48kHz → 16kHz)
```

```
┌──────────────────────────────────────────────────────────────────────┐
│ Uses `rubato` SincFixedIn resampler (high quality)                   │
│ Why? Whisper was trained on 16kHz audio                              │
└──────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 4: 📦 BUFFER INTO 6-SECOND CHUNKS
┌──────────────────────────────────────────────────────────────────────┐
│ Accumulate until: buffer.len() >= 96,000 samples                     │
│ Then: Extract chunk, continue buffering                              │
└──────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 5: 🔇 VAD CHECK (Voice Activity Detection)
┌──────────────────────────────────────────────────────────────────────┐
│ Calculate RMS (Root Mean Square) "loudness"                          │
│ If RMS < 0.005 → Skip (silence)                                      │
│ If RMS > 0.005 → Process (speech detected)                           │
└──────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 6: 🧠 WHISPER AI TRANSCRIPTION
┌──────────────────────────────────────────────────────────────────────┐
│ model.forward(audio_chunk) → "Hello world"                           │
│ Processing time: ~150ms on GPU (40x realtime!)                       │
└──────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 7: 💾 CUMULATIVE CONTEXT
┌──────────────────────────────────────────────────────────────────────┐
│ Save transcript for next chunk (last_transcript field)               │
│ Helps AI understand: "He said" → who is "he"?                        │
└──────────────────────────────────────────────────────────────────────┘
         │
         ▼
STEP 8: 📤 SEND TO UI
┌──────────────────────────────────────────────────────────────────────┐
│ emit("transcription-chunk", { text, method: "Whisper" })             │
└──────────────────────────────────────────────────────────────────────┘

═══════════════════════════════════════════════════════════════════════
```

⚠️ Gotcha: Why 6-Second Chunks?

Common Mistake: "Why not 1-second chunks for faster updates?"

Answer:

- Too short (1-2s) → Cuts words mid-sentence → AI "hallucinates" (makes up text)
- Too long (30s+) → High latency → Feels slow
- **6 seconds** → Sweet spot: complete sentences + reasonable latency

🔇 Voice Activity Detection (VAD)

🚥 Traffic Light Analogy

```
┌─────────────────────────────────────────────────────────────────┐
│                    🚥 VAD = TRAFFIC LIGHT                       │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  AUDIO CHUNK ARRIVES                                            │
│         │                                                       │
│         ▼                                                       │
│    ┌─────────┐                                                  │
│    │   VAD   │                                                  │
│    │  Check  │                                                  │
│    └────┬────┘                                                  │
│         │                                                       │
│    ┌────┴────┐                                                  │
│    │         │                                                  │
│    ▼         ▼                                                  │
│  🟢 GREEN   🔴 RED                                              │
│  Speech!   Silence.                                             │
│    │         │                                                  │
│    ▼         ▼                                                  │
│  PROCESS   SKIP                                                 │
│  with AI   (save CPU)                                           │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

🧮 How VAD Works (Energy-Based)

```rust
// Simplified VAD logic from vad.rs
fn is_speech(audio: &[f32]) -> bool {
    // Calculate RMS (Root Mean Square) - a measure of "loudness"
    let sum_squares: f32 = audio.iter().map(|s| s * s).sum();
    let rms = (sum_squares / audio.len() as f32).sqrt();

    // Compare to threshold
    rms > 0.005  // Returns true if louder than threshold
}
```

📊 VAD Benefits

Feature	Without VAD	With VAD	Benefit
CPU Load	Constant	Low during pauses	Cooler system
Final Speed	~1000ms	~550ms	**45% Faster**
Accuracy	May hallucinate	Clean silence	No phantom text

⚠️ Gotcha: VAD Threshold

Common Mistake: "VAD keeps marking my speech as silence!"

Solution: The threshold (0.005) might be too high for quiet speakers. You can:

1. Increase microphone volume in system settings
2. Speak closer to the microphone
3. (Advanced) Adjust the threshold in `vad.rs`

🧠 LLM Integration: Grammar Correction

📝 Editor Analogy

```
┌─────────────────────────────────────────────────────────────────┐
│                    📝 LLM = PERSONAL EDITOR                        │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  INPUT (from transcription):                                    │
│  "the quick brown fox jump over the lazy dog"                   │
│                     │                                           │
│                     ▼                                           │
│                                                                 │
│  ┌──────────────────────────────────────────────┐               │
│  │     Qwen 2.5 0.5B Instruct (fine-tuned)      │               │
│  │         Q4_K_M GGUF quantized                │               │
│  │                                               │               │
│  │  System: "You are Wispr Flow, an AI that      │               │
│  │           transcribes and polishes speech.    │               │
│  │           Style: Professional"               │               │
│  │                                               │               │
│  │  User: "the quick brown fox jump over..."    │               │
│  │                                               │               │
│  │  Assistant: "The quick brown fox jumps       │               │
│  │              over the lazy dog."             │               │
│  └──────────────────────────────────────────────┘               │
│                     │                                           │
│                     ▼                                           │
│  OUTPUT:                                                        │
│  "The quick brown fox jumps over the lazy dog."                 │
│                                                                 │
│  ☑ Fixed: Capitalization, subject-verb agreement                 │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

📁 LLM Files Required

```
taurscribe-runtime/models/qwen_finetuned_gguf/
└── model_q4_k_m.gguf    ← Fine-tuned Qwen 2.5 0.5B weights (~400 MB)
```

> **Note**: The LLM path is resolved in `llm.rs → get_grammar_llm_dir()`:
>
> 1. **Hardcoded absolute path** — `GRAMMAR_LLM_PATH` const at the top of `llm.rs` (points to the developer's local machine path; update when deploying)
> 2. Falls back to `GRAMMAR_LLM_DIR` environment variable
> 3. Final fallback: `%LOCALAPPDATA%\Taurscribe\models\qwen_finetuned_gguf\`

🔄 LLM Processing Flow

```
Text Input │
           ▼
┌──────────────────────┐
│ Build ChatML Prompt  │  ← "<|im_start|>system\nYou are Wispr Flow...<|im_end|>"
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ Clear KV Cache       │  ← CRITICAL: Prevents "inconsistent sequence" errors
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ Tokenize             │  ← "Hello wrold" → [token_ids...]
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ Prefill Batch        │  ← Process all prompt tokens at once (fast)
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ Decode Loop          │  ← Generate one token at a time
│ (Temperature 0.3)    │     Temp=0.3 means more deterministic output
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ Stop When EOS token  │  ← Stops when <|im_end|> or EOS token found
│ or max_tokens hit    │     max_tokens = (text.len() / 2) + 128 (dynamic)
└──────────┬───────────┘
           ▼
┌──────────────────────┐
│ Decode to Text       │  ← token_ids → "The quick brown fox..."
└──────────┬───────────┘
           ▼
Corrected Text Output
```

🎨 Transcription Styles

The LLM supports 6 styles selectable from the **LLM & Grammar** settings tab:

Style	What it does
Auto	Default — clean and natural

Style	What it does
Casual	Relaxed tone, contractions kept
Verbatim	Minimal changes, preserves original phrasing
Enthusiastic	Energetic tone, exclamation marks
Software Dev	Preserves technical terms, camelCase, CLI flags
Professional	Formal grammar, business-ready

⚠️ Gotcha: LLM KV Cache Must Be Cleared

Common Mistake: "The LLM crashes after the second transcription!"

Answer: Each new request **must** call `ctx.clear_kv_cache_seq(None)` before filling the batch. Without this, llama.cpp throws a sequence inconsistency error and panics.

⚠️ Gotcha: LLM Backend Selection

The **Auto / GPU** option sets `n_gpu_layers = 99` (offloads all layers to GPU). If GPU loading fails, it automatically retries with `n_gpu_layers = 0` (CPU only). On macOS, GPU is always forced off regardless of selection.

📝 Spell Checking

🔤 Dictionary Analogy

```
┌─────────────────────────────────────────────────────────────────┐
│                     🔤 SPELL CHECK FLOW                          │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  Input: "The quck brown fox"                                    │
│                │                                                │
│                ▼                                                │
│  ┌───────────────────────────────────────┐                      │
│  │ For each word:                        │                      │
│  │   "The"   → Found in dictionary ✓     │                      │
│  │   "quck"  → NOT FOUND! ❌              │                      │
│  │   "brown" → Found in dictionary ✓     │                      │
│  │   "fox"   → Found in dictionary ✓     │                      │
│  └───────────────────────────────────────┘                      │
│                │                                                │
│                ▼                                                │
│  ┌───────────────────────────────────────┐                      │
│  │ Find similar words to "quck":         │                      │
│  │   "quick" (edit distance: 1) ← BEST   │                      │
│  │   "duck"  (edit distance: 1)          │                      │
│  │   "quack" (edit distance: 2)          │                      │
│  └───────────────────────────────────────┘                      │
│                │                                                │
│                 ▼                                                │
│  Output: "The quick brown fox"                                   │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

📁 Spell Check Implementation

File: `src-tauri/src/spellcheck.rs`
Commands: `src-tauri/src/commands/spellcheck.rs`

The spell checker uses **SymSpell** (frequency-based edit distance):

1. Loads a frequency dictionary (`frequency_dictionary_en_82_765.txt`)
2. Splits input text into words
3. For each unknown word, finds closest matches by edit distance
4. Auto-corrects based on word frequency ranking

Dictionary location: `%LOCALAPPDATA%\Taurscribe\models\symspell\`

⚠️ Gotcha: Technical Terms

Common Mistake: "It keeps marking my technical terms as misspelled!"

Solution: Technical terms (like "ONNX", "CUDA", "API") may not be in the dictionary. SymSpell is conservative — it won't auto-correct a word if no close match exists.

📥 Model Downloads

📦 Package Delivery Analogy

```
┌──────────────────────────────────────────────────────────────────┐
│                    📦 MODEL DOWNLOAD SYSTEM                         │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  1️⃣ USER REQUEST                                                   │
│     Click "Download" in the Downloads tab                        │
│                 │                                                │
│                 ▼                                                │
│  2️⃣ FRONTEND → invoke("download_model", { modelId })               │
│     Looks up config in commands/model_registry.rs                │
│                 │                                                │
│                 ▼                                                │
│  3️⃣ DOWNLOAD MANAGER (commands/downloader.rs)                      │
│     • Fetches file(s) from Hugging Face CDN                      │
│     • Streams bytes to disk with progress                        │
│     • Emits "download-progress" events to UI                     │
│                 │                                                │
│                 ▼                                                │
│  4️⃣ VERIFICATION                                                   │
│     • SHA-1 hash checked against model_registry.rs              │
│      • File deleted if hash mismatch (corrupted)          │
│                  │                                        │
│                  ▼                                        │
│  5️⃣ COMPLETION                                            │
│      • emit("download-progress", { status: "done" })      │
│      • Frontend refreshes model list                      │
│      • Model instantly available for use!                 │
│                                                           │
└───────────────────────────────────────────────────────────┘
```

📁 Available Models (from `commands/model_registry.rs`)

Model ID	Type	Files	Size
`whisper-tiny`	Whisper GGML	1 `.bin`	~75 MB
`whisper-tiny-q5_1`	Whisper GGML (quantized)	1 `.bin`	~30 MB
`whisper-base`	Whisper GGML	1 `.bin`	~142 MB
`whisper-base-en`	Whisper GGML	1 `.bin`	~142 MB
`whisper-small`	Whisper GGML	1 `.bin`	~466 MB
`whisper-small-en`	Whisper GGML	1 `.bin`	~466 MB
`whisper-medium`	Whisper GGML	1 `.bin`	~1.5 GB
`whisper-large-v3`	Whisper GGML	1 `.bin`	~2.9 GB
`whisper-large-v3-turbo`	Whisper GGML	1 `.bin`	~1.6 GB
`parakeet-nemotron`	Parakeet ONNX	4 files	~700 MB
`qwen2.5-0.5b-instruct`	GGUF	1 `.gguf`	~400 MB
`qwen2.5-0.5b-instruct-tokenizer`	Tokenizer JSON files	4 files	~2 MB
`qwen2.5-0.5b-safetensors`	SafeTensors (GPU)	multi-file	~1 GB
`symspell-en-82k`	Dictionary	1 `.txt`	~6 MB

⚠️ Gotcha: Download Verification

Common Mistake: "The model downloaded but won't load!"

Answer: The download might be corrupted. The downloader:

1. Checks SHA-1 hash after download (hash stored in `model_registry.rs`)
2. Deletes the file if hash doesn't match
3. You'll see an error toast if verification fails

Try re-downloading or check your internet connection.

Rust Basics You Need to Know

🧩 Ownership Puzzle Analogy

```
┌─────────────────────────────────────────────────────────────────┐
│                    🧩 RUST OWNERSHIP RULES                        │
├─────────────────────────────────────────────────────────────────┤
│                                                                 │
│  Rule 1: Each value has ONE owner                               │
│  ─────────────────────────────────                               │
│  let s1 = String::from("hello");                                │
│  let s2 = s1;  // s1 is MOVED to s2                             │
│  // println!("{}", s1);  ← ERROR! s1 no longer valid          │
│                                                                 │
│  Rule 2: When owner goes out of scope, value is dropped         │
│  ──────────────────────────────────────────────────────          │
│  {                                                               │
│      let s = String::from("hello");                             │
│      // s is valid here                                          │
│  }  // s is dropped here (memory freed)                         │
│                                                                 │
│  Rule 3: You can BORROW with references                         │
│  ──────────────────────────────────────                          │
│  fn print_length(s: &String) {  // Borrows, doesn't own        │
│      println!("{}", s.len());                                    │
│  }                                                               │
│  let s = String::from("hello");                                 │
│  print_length(&s);  // Borrow                                    │
│  println!("{}", s);  // Still valid! ✓                         │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

📝 Quick Reference Table

Concept	Syntax	Example
Variable	`let x = 5;`	`let name = "Rust";`
Mutable	`let mut x = 5;`	`let mut counter = 0;`
Reference	`&x`	`let ref = &value;`
Mutable ref	`&mut x`	`let mut_ref = &mut value;`
Option	`Option<T>`	`Some(5)` or `None`
Result	`Result<T, E>`	`Ok(5)` or `Err("error")`
Match	`match x { ... }`	Pattern matching
If let	`if let Some(x) = opt { }`	Pattern matching shortcut
Unwrap	`x.unwrap()`	Get value or panic

Concept	Syntax	Example
Question mark	`x?`	Propagate error

🔒 How Shared State Works (`Arc<Mutex<T>>`)

Taurscribe shares engines (Whisper, Parakeet, LLM) across threads safely:

```rust
// In state.rs — wrapping the WhisperManager for thread-safe sharing
pub whisper: Arc<Mutex<WhisperManager>>,
//           ^^^  ^^^^^
//           │     └── Mutual Exclusion: only one thread at a time
//           └── Atomic Reference Count: multiple owners across threads

// In commands/recording.rs — using it from a background thread
let whisper = Arc::clone(&state.whisper);
std::thread::spawn(move || {
    let mut w = whisper.lock().unwrap(); // Lock, then use
    w.transcribe_chunk(&audio)?;
});
```

⚠ Gotcha: `unwrap()` is Dangerous!

Common Mistake: Using `unwrap()` everywhere

Problem: `unwrap()` panics if the value is `None` or `Err`, crashing your app!

Solution: Use safer alternatives:

```rust
// ❌ Bad
let value = maybe.unwrap();  // Crashes if None!

// ✅ Good - provide default
let value = maybe.unwrap_or(0);

// ✅ Good - handle both cases
if let Some(v) = maybe {
    println!("Got: {}", v);
}

// ✅ Good - propagate error
let value = maybe.ok_or("No value")?;
```

Complete Flow: Start to Finish

🗓 Phase 1: User Clicks "Start Recording"

```
┌─────────────────────────────────────────────────────────────────────┐
│  FRONTEND (useRecording.ts + App.tsx)                                │
├─────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  handleStartRecording() {                                            │
│      1. Check engine is loaded (Whisper or Parakeet)               │
│      2. invoke("start_recording")  →  Backend                       │
│      3. Set UI state to "Recording"                                  │
│      4. Update tray icon via invoke("set_tray_state")              │
│  }                                                                   │
│                                                                      │
│  BACKEND (commands/recording.rs)                                     │
│                                                                      │
│  pub fn start_recording(state: State<AudioState>) {                  │
│      1. Clear engine context (last_transcript = "")                │
│      2. Open default microphone (cpal)                              │
│      3. Create WAV file writer (hound)                              │
│      4. Create channels: file_tx, transcriber_tx                  │
│      5. Spawn writer_thread   → saves audio to disk                 │
│      6. Spawn transcriber_thread → real-time AI inference           │
│      7. Start audio stream (calls callback every ~10ms)             │
│  }                                                                   │
│                                                                      │
└─────────────────────────────────────────────────────────────────────┘
```

🎤 Phase 2: Audio Capture (Every ~10ms)

```
┌─────────────────────────────────────────────────────────────────────┐
│  AUDIO CALLBACK (runs on CPAL audio thread)                          │
├─────────────────────────────────────────────────────────────────────┤
│                                                                      │
│  move |data: &[f32], _| {                                            │
│                                                                      │
│      // 1. Send raw stereo to file writer                            │
│      file_tx.send(data.to_vec()).ok();                               │
│                                                                      │
│      // 2. Convert stereo → mono                                     │
│      let mono = data.chunks(2)                                       │
│                     .map(|c| (c[0] + c[1]) / 2.0)                  │
│                     .collect();                                      │
│                                                                      │
│      // 3. Send mono to transcription thread                         │
│      transcriber_tx.send(mono).ok();                                 │
│  }                                                                   │
│                                                                      │
└─────────────────────────────────────────────────────────────────────┘
```

🧠 Phase 3: Transcription Thread Loop

```
┌─────────────────────────────────────────────────────────────────┐
│  TRANSCRIPTION THREAD (background thread)                        │
├─────────────────────────────────────────────────────────────────┤
│  loop {                                                          │
│      // 1. Receive mono audio samples from channel               │
│      let samples = rx.recv()?;  // Blocks until data arrives    │
│                                                                  │
│      // 2. Add to ring buffer                                    │
│      buffer.extend(samples);                                     │
│                                                                  │
│      // 3. Check if buffer is large enough                       │
│      if buffer.len() >= chunk_size {   // 96k for Whisper        │
│          let chunk = buffer.drain(..chunk_size).collect();       │
│                                                                  │
│          // 4. [Whisper only] Skip silence with VAD              │
│          if engine == Whisper && !vad.is_speech(&chunk) {        │
│              continue; // skip this chunk                        │
│          }                                                       │
│                                                                  │
│          // 5. Resample to 16kHz                                  │
│          let resampled = resample_to_16k(&chunk);               │
│                                                                  │
│          // 6. Transcribe with AI engine                         │
│          let text = engine.transcribe_chunk(&resampled)?;        │
│                                                                  │
│          // 7. Emit live result to frontend                       │
│          app.emit("transcription-chunk", &text);                 │
│      }                                                           │
│  }  // Loop ends when channel is dropped (recording stopped)     │
│                                                                  │
└─────────────────────────────────────────────────────────────────┘
```

🛑 Phase 4: Stop Recording + Post-Processing

```
┌─────────────────────────────────────────────────────────────────┐
│  STOP RECORDING (Frontend → Backend → Frontend)                  │
├─────────────────────────────────────────────────────────────────┤
│                                                                  │
│  FRONTEND (useRecording.ts):                                     │
│  1. invoke("stop_recording") → gets raw transcript back         │
│                                                                  │
│  BACKEND (commands/recording.rs):                                │
│  2. drop(file_tx)        → signals writer thread to finish      │
│  3. writer_thread.join() → waits for WAV file to finalize       │
│  4. [Whisper] Final pass on full WAV file (higher accuracy)     │
│  5. [Parakeet] Returns accumulated session transcript           │
│  6. clean_transcript()   → fixes spacing, punctuation          │
│  7. Returns final text to frontend                               │
│                                                                  │
│  FRONTEND post-processing pipeline:                              │
│  8.  [if spell check ON] → invoke("correct_spelling")        │
│  9.  [if grammar LLM ON] → invoke("correct_text", { style })  │
│  10. invoke("type_text") → Enigo types text into active window │
│  11. Update UI transcript display                               │
│  12. Update tray icon back to "Ready"                           │
│                                                                 │
└─────────────────────────────────────────────────────────────────┘
```

⚠️ Gotcha: Channel Closing

Common Mistake: "The app hangs when I stop recording!"

How channels work:

1. `drop(file_tx)` closes the **sending end** of the channel
2. The writer thread's `rx.recv()` returns `Err` when the sender is gone
3. Thread exits its loop and finalizes the WAV file
4. **Without `drop()`**, the thread would block forever waiting for data!

📐 Module Architecture

🗂️ Current File Structure (Updated February 2026)

```
Taurscribe/
├── 🎨 Frontend
│   ├── src/
│   │   ├── App.tsx                  # UI assembly + event wiring (~440 lines)
│   │   ├── App.css                  # App-level styling
│   │   ├── main.tsx                 # React entry point
│   │   │
│   │   ├── hooks/                   # All logic lives here — App.tsx just assembles
│   │   │   ├── useHeaderStatus.ts    # Transient status ticker messages (~25 lines)
│   │   │   ├── useModels.ts          # Whisper + Parakeet model lists (~55 lines)
│   │   │   ├── usePostProcessing.ts  # LLM + SymSpell toggle/auto-load (~94 lines)
│   │   │   ├── useEngineSwitch.ts    # Engine switching + model loading (~193 lines)
│   │   │   └── useRecording.ts       # Record start/stop + post-processing (~197 lines)
│   │   │
│   │   └── components/
│   │       ├── SettingsModal.tsx      # Modal shell + tab router (~357 lines)
│   │       ├── SettingsModal.css      # Modal styling
│   │       └── settings/             # Settings tab sub-components
│   │           ├── GeneralTab.tsx     # Spell check toggle tab (~90 lines)
│   │           ├── DownloadsTab.tsx   # Model download list tab (~120 lines)
│   │           ├── ModelRow.tsx       # Single downloadable model row (~130 lines)
│   │           └── types.ts           # Shared types + MODELS constant (~125
lines)
│   │
│   └── index.html                    # HTML shell
│
├── 🦀 Backend (Rust)
│   └── src-tauri/
│       ├── src/
│       │   ├── 🎯 Core
│       │   │   ├── lib.rs               # App entry + module declarations (~132
lines)
│       │   │   ├── main.rs              # Binary entry point (6 lines)
│       │   │   ├── types.rs             # Shared enums: AppState, ASREngine (~30
lines)
│       │   │   ├── state.rs             # AudioState struct + new() (~68 lines)
│       │   │   ├── utils.rs             # get_models_dir(), get_recordings_dir(),
│       │   │   │                       # clean_transcript() (~64 lines)
│       │   │   └── audio.rs             # RecordingHandle struct (~24 lines)
│       │   │
│       │   ├── 🎤 Audio & ASR Engines
│       │   │   ├── whisper.rs           # WhisperManager: load, transcribe,
resample
│       │   │   │                       # GPU detection (CUDA→Vulkan→CPU) (~630
lines)
│       │   │   ├── parakeet.rs          # ParakeetManager: Nemotron/CTC/EOU/TDT
│       │   │   │                       # transcription + model status (~339
lines)
│       │   │   ├── parakeet_loaders.rs # GPU/CPU loader helpers for each
│       │   │   │                       # Parakeet model type (~300 lines)
│       │   │   └── vad.rs               # Energy-based VAD: is_speech(),
│       │   │                           # get_speech_timestamps() (~162 lines)
│       │   │
│       │   ├── ✨ Post-Processing
│       │   │   ├── llm.rs               # LLMEngine: Qwen 2.5 0.5B GGUF via
│       │   │   │                       # llama-cpp-2, format_transcript() (~343
lines)
│       │   │   └── spellcheck.rs        # SymSpell spell checker (~150 lines)
│       │   │
│       │   ├── 📡 Commands (Tauri IPC)
│       │   │   └── commands/
│       │   │       ├── mod.rs              # Re-exports all pub commands
│       │   │       ├── recording.rs        # start_recording, stop_recording,
type_text
│       │   │       ├── models.rs           # list_models, switch_model,
init_parakeet,
│       │   │       │                       # set_active_engine, get_backend_info,
etc.
│       │   │       ├── llm.rs              # init_llm, unload_llm, correct_text,
│       │   │       │                       # check_llm_status
│       │   │       ├── spellcheck.rs       # init_spellcheck, unload_spellcheck,
│       │   │       │                       # correct_spelling,
check_spellcheck_status
│       │   │       ├── downloader.rs       # download_model, get_download_status,
│       │   │       │                       # delete_model, verify_model_hash
│       │   │       ├── model_registry.rs   # get_model_config(): all model URLs +
SHA1s
│       │   │       ├── settings.rs         # set_tray_state
│       │   │       └── misc.rs             # greet (placeholder)
│       │   │
│       │   ├── 🖼️ System Tray
│       │   │   └── tray/
│       │   │       ├── mod.rs              # setup_tray() + icon switching
│       │   │       └── (icon assets)
│       │   │
│       │   ├── ⌨️ Global Hotkeys
│       │   │   └── hotkeys/
│       │   │       ├── mod.rs              # Re-exports start_hotkey_listener
│       │   │       └── listener.rs         # rdev Ctrl+Win listener (~75 lines)
│       │   │
│       │   └── 👁️ File Watcher
│       │       └── watcher.rs              # notify watcher on models dir,
│       │                                   # emits "models-changed" event (~60
lines)
│       │
│       ├── build.rs              # macOS deployment target, CUDA linker paths
│       └── Cargo.toml            # All Rust dependencies + feature flags
│
├── 📦 Runtime Assets
│   └── taurscribe-runtime/
│       └── models/
│           ├── qwen_finetuned_gguf/  # model_q4_k_m.gguf
│           └── parakeet-*/           # ONNX model folders (dev only)
│
├── assets/                       # App icons, tray icons (.png / .icns / .ico)
│
└── 📚 Documentation
    ├── ARCHITECTURE.md           # This file!
    └── README.md
```

🏗️ Module Dependency Diagram

```
┌─────────────────────────────────────────────────────────┐
│                lib.rs  (top level)                      │  ← Declares all
modules
├─────────────────────────────────────────────────────────┤
│  commands/   tray/   hotkeys/   watcher                 │  ← Feature modules
├─────────────────────────────────────────────────────────┤
│  whisper   parakeet   vad   llm   spellcheck            │  ← AI engines
│                │                                        │
│         parakeet_loaders                                │  ← Loader helpers
├─────────────────────────────────────────────────────────┤
│  commands/model_registry   commands/downloader          │  ← Download subsystem
│  (registry has no deps)    (uses registry + utils)      │
├─────────────────────────────────────────────────────────┤
│  types   state   utils   audio                          │  ← Core (no
dependencies)
└──────────────────────────────────────────────────────────┘


Rule: Lower modules NEVER depend on higher modules!

Frontend hook dependency order:
  useHeaderStatus   ←  (no deps)
  useModels         ←  useHeaderStatus
  usePostProcessing ←  useHeaderStatus
  useEngineSwitch   ←  useModels, useHeaderStatus
  useRecording      ←  useEngineSwitch, usePostProcessing, useHeaderStatus
  App.tsx           ←  all hooks
```

⚠️ Gotcha: Circular Dependencies

Common Mistake: "I added `use crate::commands` to `whisper.rs` and it won't compile!"

Solution: Lower-level modules (`whisper.rs`, `llm.rs`) must NEVER import from higher-level modules (`commands/`). Instead:

- Put shared types in `types.rs`
- Put utility functions in `utils.rs`
- Let the higher-level module (commands) import from the lower-level ones

---

🔬 Deep Dives: How the Tricky Code Actually Works

> These sections break down the most confusing or "magic-looking" parts of the codebase into the simplest possible explanations. Each example is taken directly from the real code.

1️⃣ Channels — Threads Talking to Each Other

In `commands/recording.rs`, the code uses **channels** to send audio from the microphone thread to the transcription thread. Think of a channel exactly like a **walkie-talkie**:

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                    📻 HOW CHANNELS WORK                                        │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                               │
│   SENDER end                               RECEIVER end                      │
│   (tx = "transmitter")                     (rx = "receiver")                │
│                                                                               │
│   🎤 Audio Thread             channel pipe          🧠 Transcription Thread   │
│   ┌──────────────┐          ═══════════════          ┌──────────────────┐    │
│   │              │──────── [data] [data] ───────────►│                  │    │
│   │  tx.send()   │          (queue of data)          │   rx.recv()      │    │
│   │  audio data  │                                   │ waits here       │    │
│   └──────────────┘                                   └──────────────────┘    │
│                                                                               │
│   Key rules:                                                                  │
│   • tx.send(data) → puts data into the pipe (never blocks)           │
│   • rx.recv()     → takes data OUT (BLOCKS until data arrives)       │
│   • drop(tx)      → closes pipe → rx.recv() returns Err → thread exits     │
│                                                                        │
└──────────────────────────────────────────────────────────────────────┘
```

Annotated real code from `commands/recording.rs`:

```
// Step 1: Create TWO channels — one for file writing, one for transcription
let (file_tx, file_rx) = crossbeam_channel::unbounded::<Vec<f32>>();
//    ^^^^^^  ^^^^^^^                                   ^^^^^^^^^^
//    Sender  Receiver                                  Any size queue (no limit)

let (transcriber_tx, transcriber_rx) = crossbeam_channel::unbounded::<Vec<f32>>();

// Step 2: Spawn a background thread that owns the RECEIVER end
std::thread::spawn(move || {
//                 ^^^^
//                 "move" = this thread now OWNS transcriber_rx
    loop {
        match transcriber_rx.recv() {   // ← BLOCKS here, waiting for audio data
            Ok(samples) => { /* transcribe */ }
            Err(_) => break,             // ← tx was dropped = recording stopped
        }
    }
});

// Step 3: Audio callback runs on CPAL's thread (every ~10ms)
let callback = move |data: &[f32], _: &_| {
    file_tx.send(data.to_vec()).ok();         // → file writer thread
    transcriber_tx.send(data.to_vec()).ok();  // → transcription thread
    //                                  ^^^^
    //                   .ok() = ignore send error if receiver is gone
};

// Step 4: When recording stops, drop the sender → threads finish naturally
drop(file_tx);            // ← File writer thread sees Err and exits
drop(transcriber_tx);     // ← Transcription thread sees Err and exits
writer_thread.join().unwrap();  // ← Wait for both to finish cleanly
```

> ⚠ **Gotcha — Why `move` before the closure?**
> Without `move`, the closure would borrow `transcriber_rx` by reference. But references can't cross thread boundaries in Rust (the original thread might die first). The `move` keyword transfers **ownership** into the new thread, making it safe.

---

2 `Arc<Mutex<T>>` — Sharing a Resource Between Threads

Taurscribe's AI engines (Whisper, Parakeet, LLM) live in `state.rs` and need to be accessed from *multiple* threads. Here's how that works:

```
┌──────────────────────────────────────────────────────────────────────────────┐
│               🏦  Arc<Mutex<T>> = Thread-Safe Safe Deposit Box                 │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                                │
│   Arc (Atomic Reference Count)                                                 │
│   ───────────────────────────                                                  │
│   Imagine a "shared photocopy" of a key.                                       │
│   You can make as many copies as you need.                                     │
│   The box is destroyed only when ALL copies are gone.                          │
│                                                                                │
│   ┌──────────────────────────────────────────────────────────────────┐         │
│   │ Original Arc  ──── copy 1 (Thread A: recording command)          │         │
│   │               └─── copy 2 (Thread B: transcription thread)       │         │
│   │               └─── copy 3 (Thread C: stop command)               │         │
│   │                                                                    │         │
│   │  ref count: 3  →  box still alive                                │         │
│   └──────────────────────────────────────────────────────────────────┘         │
│                                                                                │
│   Mutex (Mutual Exclusion)                                                     │
│   ─────────────────────────                                                    │
│   Only ONE thread can look inside the box at a time.                           │
│   Others must wait outside until the door opens.                               │
│                                                                                │
│   Thread A ──► lock() ──► USES WhisperManager ──► DROP (auto-unlock)         │
│   Thread B ──► lock() ──► [WAITING...] ─────────────────────────────►        │
│                           (waits until Thread A is done)                       │
│                                                                                │
└──────────────────────────────────────────────────────────────────────────────┘
```

Annotated real code from `state.rs` and `commands/recording.rs`:

```
// In state.rs — declaring the shared state
pub struct AudioState {
    pub whisper: Arc<Mutex<WhisperManager>>,
    //           ^^^  ^^^^^
    //           │     └── "One thread at a time" lock
    //           └── "Multiple owners" reference-counted pointer

    pub parakeet: Arc<Mutex<ParakeetManager>>,
    pub llm:      Arc<Mutex<LLMEngine>>,
}

// In commands/recording.rs — using WhisperManager from a background thread
pub fn start_recording(state: tauri::State<AudioState>) {

    // Clone the Arc (cheap! just increments the reference count)
    let whisper_clone = Arc::clone(&state.whisper);
    //                             ^^^^^^^^^^^^^^^
    //                             borrow to clone — doesn't move the original
    std::thread::spawn(move || {   // Move the clone INTO the thread

        // Lock the mutex — we now have exclusive access
        let mut w = whisper_clone.lock().unwrap();
        //          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        //          Returns MutexGuard<WhisperManager>
        //          Auto-unlocks when `w` goes out of scope (RAII)

        w.transcribe_chunk(&audio)?;  // Use WhisperManager safely

    }); // `w` drops here = mutex UNLOCKED = other threads can now use it
}
```

> ⚠ **Gotcha — Deadlock!**
> What if Thread A locks `whisper`, then tries to lock `llm`, while Thread B has `llm` locked and waits for `whisper`? Both threads wait forever — **deadlock**!
> **Rule**: Always lock mutexes in the same order everywhere in the code.

---

3 The Tauri IPC Bridge — JavaScript Calling Rust

This is the "magic" that lets the React UI call Rust functions:

```
┌───────────────────────────────────────────────────────────────────────────────┐
│              🌉 THE TAURI IPC BRIDGE                                            │
├───────────────────────────────────────────────────────────────────────────────┤
│                                                                                │
│  FRONTEND (TypeScript / React)            BACKEND (Rust)                       │
│  ─────────────────────────────            ───────────────                      │
│                                                                                │
│  // 1. Call a Rust function              // 3. Rust function receives it       │
│  const text = await invoke(              #[tauri::command]                     │
│    "correct_text",           ═══════════►pub fn correct_text(                  │
│    { text: "hello wrold",                    text: String,                     │
│      style: "Professional" }                 style: String,                    │
│  );                                          state: State<AudioState>          │
│  // 4. Rust return value flows back     ) -> Result<String, String> {          │
│  console.log(text);          ◄════════════   // ... LLM correction ...         │
│  // "Hello world."                           Ok(corrected_text)                │
│                                          }                                     │
│                                                                                │
│  // 2. Event: Rust PUSHES to frontend    // 5. Rust emits an event             │
│  listen("transcription-chunk",           app_handle.emit(                      │
│    (event) => {              ◄════════════   "transcription-chunk",            │
│      showText(event.payload)                 &chunk_text                       │
│    }                                     );                                    │
│  );                                                                            │
│                                                                                │
│  invoke = SYNCHRONOUS request/response (you await the answer)                  │
│  listen = ASYNCHRONOUS subscription   (Rust fires whenever it wants)          │
│                                                                                │
└──────────────────────────────────────────────────────────────────────────────┘
```

How a command gets registered — `lib.rs`:

```rust
// lib.rs — this is like a phone directory: "here are all the functions
//           the frontend is allowed to call"
tauri::Builder::default()
    .invoke_handler(tauri::generate_handler![
        //                   ^^^^^^^^^^^^^^^^^
        //                   Macro that wires up the IPC handler
        commands::start_recording,  // JS can call invoke("start_recording")
        commands::stop_recording,   // JS can call invoke("stop_recording")
        commands::correct_text,     // JS can call invoke("correct_text", {text, style})
        // ... etc for every command
    ])
```

> ⚠ **Gotcha — Naming matters!**
> The string you pass to `invoke("start_recording")` in JavaScript must **exactly** match the Rust function name. A typo gives a silent runtime error, not a compile error.

4️⃣ `Result<T, E>` and the `?` Operator — Rust Error Handling

Rust has no exceptions. Instead, functions return `Result<Ok, Err>` — a box that contains *either* a success value or an error:

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                  📦 Result<T, E> = A Box With Two Compartments                  │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                                │
│    Result<String, String>                                                      │
│    ┌──────────────────────────────────────────────┐                            │
│    │                                              │                            │
│    │   Compartment A: Ok(String)                  │                            │
│    │    ✅ "The corrected text result"              │
│                                                                                
│    │                                              │                            │
│    │   Compartment B: Err(String)                 │                            │
│    │    ❌ "Model not loaded: file not found"   │                                │
│    │                                              │                            │
│    └──────────────────────────────────────────────┘                            │
│                                                                                │
│                                                                                │
│    The caller MUST check which compartment has data before using it.           │
│    Rust forces this — you literally cannot use the value without checking.     │
│                                                                                │
└──────────────────────────────────────────────────────────────────────────────┘
```

Three ways to handle a `Result`:

```rust
// ────────────────────────────────────────
// WAY 1: match — explicit, handle both cases
// ────────────────────────────────────────
match load_model(path) {
    Ok(model)  => { /* use model */ }
    Err(e)     => { eprintln!("Failed: {}", e); }
}

// ────────────────────────────────────────
// WAY 2: ? operator — short-circuit on error (used EVERYWHERE)
// ────────────────────────────────────────
fn start_recording(state: State<AudioState>) -> Result<(), String> {
    let model = load_model(path)?;
    //                          ^
    //   If load_model returns Err(e), this function IMMEDIATELY
    //   returns Err(e) — no need to write the match manually.
    //   If load_model returns Ok(m), execution continues with m.

    let text = model.transcribe(&audio)?;  // Same pattern — bail on error
    Ok(())  // If we got here, everything worked!
}

// ────────────────────────────────────────
// WAY 3: unwrap_or_else — provide a default value
// ────────────────────────────────────────
let dir = get_models_dir().unwrap_or_else(|_| PathBuf::from("/tmp"));
//                                   ^
//                          If it fails, use /tmp as fallback
```

> ⚠ **Gotcha — `?` only works inside functions that return `Result`**
> If you try `let x = something()?;` inside `main()` or a closure that returns `()`, the compiler will complain. Wrap the code in a helper function that returns `Result<_, _>` first.

---

5 Thread Lifetimes — What "Spawning" a Thread Actually Means

Here is a visual timeline of how threads start and stop during a recording session:

```
┌──────────────────────────────────────────────────────────────────────────────┐
│                     ⏱ THREAD LIFETIME DURING A RECORDING                     │
├──────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  Time ──────────────────────────────────────────────────────────────────►    │
│                                                                              │
│  Main Thread ══╦═══════════════════════════════════════════╦═══════════════   │
│  (Tauri cmds)  ║ start_recording()                         ║ stop_recording() │
│                ║                                           ║                  │
│                ║ spawn ─────────────────────────────── ►   ║                  │
│  Writer Thread  ╠════════ writing WAV samples ════════════►──╢ join() → done   │
│                 ║  (owns file_rx)                            ║ → finalize WAV   │
│                 ║                                            ║                  │
│                 ║ spawn ──────────────────────────────── ►   ║                  │
│  Transcriber    ╠════════ recv → AI → emit ══════════════►──╢ join() → done   │
│  Thread         ║  (owns transcriber_rx + AI engine)         ║                  │
│                 ║                                            ║                  │
│  CPAL Thread    ╠══ audio callback (every ~10ms) ═════════►─╢ stream.stop()    │
│  (audio driver)║  [sends data via tx channels]              ║                  │
│                 ║                                            ║                  │
│         ────────── user speaks ─────────────────────────────────────────────   │
│                                                                                 │
│  Legend:  ═══════ Thread alive and running                                      │
│           ──►     Thread spawned at this point                                  │
│           ──╢     Thread receives stop signal (tx dropped)                      │
│           join()  = "wait here until that thread finishes"                      │
│                                                                                 │
└─────────────────────────────────────────────────────────────────────────────────┘
```

Why `join()` matters:

```
// Without join(): WAV file might be half-written when we return!
writing_thread.join().unwrap();
//             ^^^^^^
//             Blocks (waits) until the writing thread finishes
//             finalizing the WAV file header. THEN we return.

// The WAV format requires the file SIZE in the header.
// The writer thread fixes the header LAST, right before it exits.
// Without join(), we'd return a corrupt WAV file.
```

6️⃣ The Audio Resampling Math

Whisper requires 16,000 samples per second. Your microphone records 48,000 samples per second. Here's what resampling actually does:

```
┌─────────────────────────────────────────────────────────────────────────────────┐
│                🎵 AUDIO RESAMPLING: 48kHz → 16kHz                                │
├─────────────────────────────────────────────────────────────────────────────────┤
│                                                                                 │
│  Original (48kHz) — 48,000 numbers per second:                                  │
│  [0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, ...]                    │
│    ^─────────────────^─────────────────^                                        │
│    position 0        position 2 (→ kept)  position 4 (→ kept)                   │
│                                                                                 │
│  After resampling (16kHz) — 16,000 numbers per second:                          │
│  [0.02,              0.05,             0.08, ...]                               │
│                                                                                 │
│  The `rubato` library uses a sinc filter (math magic) to:                   │
│  • Keep every 3rd sample approximately                                        │
│  • Blend neighboring samples to avoid aliasing (audio distortion)            │
│  • Result: same audio, just 3× fewer numbers                                 │
│                                                                               │
│  Ratio: 16000 / 48000 = 1/3  →  output has 1/3 as many samples              │
│                                                                               │
└─────────────────────────────────────────────────────────────────────────────┘
```

> **Why not just take every 3rd sample directly?**
> That's called "downsampling without anti-aliasing" — it causes high-frequency audio artifacts (ugly distortion). The `rubato` sinc resampler applies a low-pass filter first to prevent this. It's like reducing a photo's resolution properly vs. just deleting every 3rd pixel.

7️⃣ The ChatML Prompt Format — How the LLM "Understands" Instructions

The LLM (Qwen 2.5) uses a special text format called **ChatML** to understand your instructions:

```
┌─────────────────────────────────────────────────────────────────────────────┐
│           📝 CHATML FORMAT — The "Protocol" the LLM Speaks                  │
├─────────────────────────────────────────────────────────────────────────────┤
│                                                                               │
│  <|im_start|>system                                                           │
│  You are Wispr Flow, an AI that transcribes and polishes speech.              │
│  Style: Professional. Fix grammar. Output ONLY the corrected text.             │
│  <|im_end|>                                                                   │
│  ──────────────────────────────────────────────────────────────────────────  │
│  ▲                                                                            │
│  │ "system" message — sets the AI's personality and rules.                   │
│    Think of it like a job description given before work starts.              │
│                                                                               │
│  <|im_start|>user                                                             │
│  the quick brown fox jump over the lazy dog                                   │
│  <|im_end|>                                                                   │
│  ──────────────────────────────────────────────────────────────────────────  │
│  ▲                                                                            │
│  │ "user" turn — this is the raw transcription text we want corrected.       │
│                                                                               │
│  <|im_start|>assistant                                                        │
│  ──────────────────────────────────────────────────────────────────────────  │
│  ▲                                                                            │
│  │ We leave this EMPTY — the model fills in the corrected text here.         │
│  │ It generates: "The quick brown fox jumps over the lazy dog."              │
│  │ Then stops when it produces <|im_end|>                                    │
│                                                                               │
└─────────────────────────────────────────────────────────────────────────────┘
```

Annotated code from `llm.rs`:

```
fn build_chatml_prompt(text: &str, style: &str) -> String {
    format!(
        // The system message — tells the AI what personality to have
        "<|im_start|>system\n\
         You are Wispr Flow, an AI assistant that transcribes speech.\n\
         Style: {style}.\n\
         Output ONLY the corrected text. No explanations.\n\
         <|im_end|>\n\
         \
         <|im_start|>user\n\
         {text}\n\
         <|im_end|>\n\
         \
         <|im_start|>assistant\n",
         //                    ^
         // No closing <|im_end|> here — the model writes everything AFTER this
        style = style,
        text  = text,
    )
}

// During inference, stop generating when we see the end token:
if token == eos_token || decoded.contains("<|im_end|>") {
    break;   // LLM is done! Collect what we have.
}
```

---

8 Closures — Anonymous Functions ("functions without a name")

Closures appear **everywhere** in Taurscribe. They look confusing at first:

```
// A normal named function:
fn add_one(x: i32) -> i32 {
    x + 1
}

// A closure (same thing, but inline and anonymous):
let add_one = |x: i32| x + 1;
//            ^^      ^
//   Parameters      Body (no curly braces needed for one expression)

// Multi-line closure:
let process = |data: Vec<f32>| {
    let mono = convert_to_mono(&data);
    resample_to_16k(&mono)
};
```

The audio callback is a closure capturing variables from the outer scope:

```
// These variables are declared OUTSIDE the closure:
let file_tx        = /* channel sender */;
let transcriber_tx = /* channel sender */;

// The closure is passed to CPAL as the audio callback.
// It "captures" file_tx and transcriber_tx from the surrounding scope.
let callback = move |data: &[f32], _info: &cpal::InputCallbackInfo| {
    //         ^^^^
    //         Moves captured variables INTO the closure
    //         (transfers ownership — outer scope can no longer use them)

    file_tx.send(data.to_vec()).ok();
    //  ^^^^^^^
    //  file_tx was "moved in" above — the closure now owns it

    transcriber_tx.send(data.to_vec()).ok();
};
// CPAL calls this closure every ~10ms on its internal audio thread
let stream = device.build_input_stream(&config, callback, err_fn, None);
```

9️⃣ Iterators and Chaining — Reading the "Fluent" Style

Rust loves chaining iterator methods. Here's how to read them:

```
// Converting stereo [L, R, L, R, ...] to mono [(L+R)/2, ...]
let mono: Vec<f32> = stereo_samples
    .chunks(2)                          // Step 1: Group into pairs [L,R], [L,R], ...
    .map(|chunk| {                      // Step 2: For each pair...
        let left  = chunk[0];           //         get left channel sample
        let right = chunk.get(1).copied().unwrap_or(left); // right (or left if missing)
        (left + right) / 2.0            //         average them → one mono sample
    })
    .collect();                         // Step 3: Gather all results into Vec<f32>
//  ^^^^^^^^^
//  Iterators are LAZY — nothing runs until collect() is called!
```

```
┌─────────────────────────────────────────────────────────────────────────────┐
│                    🔗 ITERATOR CHAIN VISUALIZATION                         │
├─────────────────────────────────────────────────────────────────────────────┤
│                                                                              │
│  Input:  [0.1, 0.3, 0.2, 0.4, 0.5, 0.7]                                    │
│  (stereo: left=0.1, right=0.3, left=0.2, right=0.4, ...)                   │
│                                                                              │
│  .chunks(2)  ──►  [0.1, 0.3]   [0.2, 0.4]   [0.5, 0.7]                    │
│                        │            │             │                          │
│  .map(avg)    ──►     0.2          0.3           0.6                         │
│                        │          │          │                       │
│  .collect()  ──►  [0.2, 0.3, 0.6]   ← mono output!                  │
│                                                                      │
└──────────────────────────────────────────────────────────────────────┘
```

🔟 `Option<T>` — When Something Might Not Exist

Many things in Taurscribe might not exist yet: the loaded model, an active recording, a found word. `Option<T>` represents "maybe a value, maybe nothing":

```
┌──────────────────────────────────────────────────────────────────────┐
│                  📦 Option<T> = A Box That Might Be Empty             │
├──────────────────────────────────────────────────────────────────────┤
│                                                                       │
│  Some(value)    ← The box HAS something inside                       │
│  None           ← The box is EMPTY                                   │
│                                                                       │
│  Real example in state.rs:                                            │
│  pub model: Option<WhisperModel>                                      │
│  //         ^^^^^^                                                    │
│  //         model might not be loaded yet!                           │
│                                                                       │
│  WRONG — crashes if None:                                             │
│  let m = state.model.unwrap();    // ❌ panics if no model loaded      │
│                                                                       │
│  RIGHT — check first:                                                 │
│  if let Some(m) = &state.model {  // ✅ safe                           │
│      m.transcribe(&audio)?;                                          │
│  } else {                                                            │
│      return Err("No model loaded".into());                           │
│  }                                                                   │
│                                                                       │
└──────────────────────────────────────────────────────────────────────┘
```

1️⃣1️⃣ The VAD Math Explained Simply (RMS)

The VAD (Voice Activity Detection) uses a formula called **RMS (Root Mean Square)**. Here's what it means in plain English:

```
┌──────────────────────────────────────────────────────────────────────┐
│                  📊 RMS FORMULA — STEP BY STEP                        │
├──────────────────────────────────────────────────────────────────────┤
│                                                                       │
│ Input audio chunk: [0.01, -0.02, 0.03, -0.01]                       │
│                                                                       │
│ Step 1: SQUARE every sample (makes all numbers positive)             │
│ [0.01²,   0.02²,   0.03²,   0.01²  ]                                │
│  [0.0001,  0.0004,  0.0009,  0.0001 ]                                      │
│                                                                              │
│  Step 2: AVERAGE the squares  (sum / count)                                 │
│  (0.0001 + 0.0004 + 0.0009 + 0.0001) / 4  =  0.000375                     │
│                                                                              │
│  Step 3: SQUARE ROOT (undo the squaring from step 1)                        │
│  √0.000375  ≈  0.019                                                        │
│                                                                              │
│  Result:    RMS = 0.019                                                     │
│  Threshold: 0.005                                                           │
│  0.019 > 0.005  →  ☑ SPEECH DETECTED                                       │
│                                                                              │
│  Intuition: RMS = "average loudness" of the audio chunk                    │
│  • Loud speech  →  high RMS  (e.g., 0.05–0.2)                              │
│  • Quiet room   →  low RMS   (e.g., 0.001–0.003)                           │
│  • Threshold 0.005 = the dividing line between speech and silence          │
│                                                                              │
└─────────────────────────────────────────────────────────────────────────────┘
```

File & Function Reference

🔍 Quick Lookup Table

I want to...	Go to	Function/Item
Add a new Tauri command	`commands/*.rs`	Add `#[tauri::command]` fn + register in `lib.rs`
Change recording behavior	`commands/recording.rs`	`start_recording()`, `stop_recording()`
Modify Whisper logic	`whisper.rs`	`transcribe_chunk()`, `WhisperManager::new()`
Modify Parakeet transcription	`parakeet.rs`	`transcribe_chunk()`, `initialize()`
Change how Parakeet loads GPU/CPU	`parakeet_loaders.rs`	`init_*_gpu()`, `init_*_cpu()`
Add a new downloadable model	`commands/model_registry.rs`	Add entry to `get_model_config()`
Add model to the downloads UI	`src/components/settings/types.ts`	Add entry to `MODELS` array
Change download/verify logic	`commands/downloader.rs`	`download_model()`, `verify_model_hash()`

I want to...	Go to	Function/Item
Change LLM prompt or style	`llm.rs`	`format_transcript()`
Change LLM inference params	`llm.rs`	`run_with_options()`
Change spell check	`spellcheck.rs`	Correction logic
Modify tray icon/behavior	`tray/mod.rs`	`setup_tray()`
Change global hotkey	`hotkeys/listener.rs`	Modify key match arms
Add shared enum/struct	`types.rs`	Define struct/enum
Add utility function	`utils.rs`	Create `pub fn`
Change UI recording logic	`src/hooks/useRecording.ts`	`handleStartRecording()`, `handleStopRecording()`
Change engine switching UI	`src/hooks/useEngineSwitch.ts`	`handleSwitchToWhisper()`, `handleSwitchToParakeet()`
Change LLM/spell UI toggles	`src/hooks/usePostProcessing.ts`	Toggle + load/unload logic
Change settings tabs	`src/components/SettingsModal.tsx`	`renderContent()`, tab list
Modify General settings tab	`src/components/settings/GeneralTab.tsx`	Spell check toggle UI
Modify Downloads tab	`src/components/settings/DownloadsTab.tsx`	Model list + ModelRow

📋 All Tauri Commands (as of February 2026)

```
// From lib.rs invoke_handler — matches tauri::generate_handler! exactly:

// 🔧 Misc
commands::greet,                   // Test/greeting placeholder

// 🎤 Recording
commands::start_recording,         // Start mic + real-time transcription
commands::stop_recording,          // Stop + final transcript + post-process
commands::type_text,               // Type text via Enigo keyboard injection

// 🧠 Whisper model management
commands::list_models,             // List downloaded Whisper .bin files
commands::get_current_model,       // Get active Whisper model name
```

```
commands::switch_model,           // Load a different Whisper model

// ⚡ Parakeet model management
commands::list_parakeet_models,   // List Parakeet models + their status
commands::init_parakeet,          // Initialize a Parakeet model (GPU/CPU)
commands::get_parakeet_status,    // Check if Parakeet is loaded + which model

// 🔀 Engine switching
commands::set_active_engine,      // Switch between Whisper / Parakeet
commands::get_active_engine,      // Get the currently active engine
commands::get_backend_info,       // Get GPU backend info string

// 🖼 System tray
commands::set_tray_state,         // Update tray icon
(Ready/Recording/Processing)

// ✨ LLM grammar correction
commands::init_llm,               // Load Qwen GGUF model (GPU or CPU)
commands::unload_llm,             // Unload LLM to free VRAM
commands::run_llm_inference,      // Raw LLM text generation
commands::check_llm_status,       // Returns bool: true = loaded, false = not
loaded
commands::correct_text,           // Format transcript with style via LLM

// 🔤 Spell checking
commands::init_spellcheck,        // Load SymSpell dictionary
commands::unload_spellcheck,      // Unload spell checker
commands::check_spellcheck_status, // Check if spell checker is loaded
commands::correct_spelling,       // Run SymSpell correction on text

// 📥 Download manager
commands::download_model,         // Stream download from Hugging Face
commands::get_download_status,    // Check downloaded/verified status per model
commands::delete_model,           // Delete model file(s) from disk
commands::verify_model_hash,      // Verify SHA-1 integrity of model file
```

Common Beginner Questions

Q1: Why are there two transcription engines?

Answer: Different use cases need different trade-offs:

- **Whisper** — Higher accuracy, 6-second latency → Best for dictation, meetings
- **Parakeet** — Lower latency (~0.6s), slightly less accurate → Best for real-time streaming

Q2: Can I use this for other languages?

Whisper supports 99 languages — just speak and it auto-detects. Parakeet is English-only (NVIDIA Nemotron model).

Q3: How much RAM does this use?

Component	RAM Usage
Whisper tiny	~100 MB
Whisper base	~200 MB
Whisper large-v3	~3 GB
Parakeet Nemotron	~500 MB
Qwen LLM (Q4_K_M)	~400 MB
Audio buffer	~10 MB

> LLM and Spell Checker are **not loaded at startup** — only when you enable them.

Q4: Why does the first transcription take longer?

Answer: GPU "warm-up"! The first run compiles CUDA/Vulkan shader kernels. Taurscribe optionally runs a warm-up pass during model initialization to hide this delay from the user.

Q5: What if my recording crashes mid-session?

Safety features:

1. WAV file is written continuously stream → disk (you don't lose audio)
2. File saved to: `%LOCALAPPDATA%\Taurscribe\temp\`
3. You can manually re-transcribe the WAV with any tool

Q6: Where do downloaded models go?

All models land in `%LOCALAPPDATA%\Taurscribe\models\`:

```
models/
├── ggml-tiny.bin               ← Whisper models
├── ggml-base.en.bin
├── parakeet-nemotron/          ← Parakeet ONNX folders
│   ├── encoder.onnx
│   └── decoder.onnx
├── qwen_finetuned_gguf/        ← Grammar LLM
│   └── model_q4_k_m.gguf
└── symspell/                   ← Spell check dictionary
    └── frequency_dictionary_en_82_765.txt
```

Q7: How does the global hotkey work?

`hotkeys/listener.rs` spawns a background thread that uses `rdev::listen()` to capture **every** key event system-wide. When both `Ctrl` + `Win (Meta)` are held:

- Sends `hotkey-start-recording` event → Frontend starts recording
- On key release → Sends `hotkey-stop-recording` → Frontend stops recording

Conclusion

Taurscribe demonstrates modern Rust practices in a real production app:

☑ **Ownership** — Threads take ownership of data they need
☑ **Borrowing** — Functions borrow without taking ownership
☑ **Concurrency** — Multiple threads work safely in parallel
☑ **Error Handling** — `Result`, `?` operator, `anyhow` for safety
☑ **Modularity** — Clean separation into focused modules after refactoring

Architecture Benefits:

Feature	Benefit
Separate threads	UI never freezes during AI inference
Crossbeam channels	Safe, backpressure-aware thread communication
`Arc<Mutex<T>>`	Shared engine state protection
Two AI engines	User picks speed OR accuracy
GPU acceleration	12–60× faster than CPU-only
`commands/` split	Each command file has one clear responsibility
`model_registry.rs`	Single source of truth for all model configs
On-demand loading	Parakeet + LLM don't use memory until needed

Key Takeaway: Rust's strict compiler prevents entire categories of bugs (data races, null pointer crashes, use-after-free). Once your 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 a new Whisper or Parakeet model variant (edit `model_registry.rs` + `types.ts`)
2. Add a new transcription style to the LLM (edit `format_transcript()` + the style dropdown)
3. Implement speaker diarization (who's speaking)
4. Add export formats (SRT, VTT, plain TXT)
5. Replace energy-based VAD with Silero neural VAD for higher accuracy

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