

Speech-to-Text Fine-Tuning Guide - Book 2 - Implementation

Speech-to-Text Fine-Tuning Guide - Book 2 - Implementation

Speech-to-Text Fine-Tuning Guide

Book 2: Implementation

Fine-Tuning, Inference & Hardware Optimization

Table of Contents

Part IV: Fine-Tuning

Fine-tuning strategies and techniques (7 chapters)

Part V: Inference & Deployment

Running and deploying ASR models (4 chapters)

Part VI: AMD GPU Optimization

AMD-specific hardware considerations (3 chapters)

Part IV: Fine-Tuning

Fine-tuning strategies and techniques

Fine Tuning Small Models Strategy

The Strategic Question

If your desktop GPU can comfortably run Whisper Small but struggles with Medium/Large, and you notice accuracy drops with stock Small compared to larger models:

Would fine-tuning Small or Tiny models be a more practical strategy than fine-tuning Large models that you can only run in the cloud?

Short Answer

Yes! Fine-tuning smaller models (Small/Tiny) for local inference is an excellent and often overlooked strategy.

The accuracy improvements from fine-tuning can be **more significant** for smaller models than larger ones, and the practical benefits for daily use are substantial:

- **Fine-tuned Whisper Small can approach or match stock Whisper Medium accuracy** for your specific voice/vocabulary
- **Fine-tuned Whisper Tiny can approach stock Small accuracy**
- You get these benefits with fast, local inference on modest hardware
- More practical than fine-tuning Large models you can only use via expensive API calls

The Math: Fine-Tuning Gains vs Model Size

Baseline Accuracy (Stock Models, General Speech)

Typical Word Error Rates (WER) on diverse audio:

Model	Parameters	WER (clean)	WER (noisy)
Large-v3	1550M	3-5%	8-12%
Medium	769M	4-7%	10-15%
Small	244M	8-12%	15-25%
Base	74M	12-18%	25-35%
Tiny	39M	15-25%	30-45%

Observation: Each size tier represents roughly 1.5-2× more errors

Fine-Tuning Improvements (Typical Gains)

When fine-tuned on 5-10 hours of personal data:

Model	Baseline WER	Fine-tuned WER	Improvement
Large-v3	5%	3-4%	1-2% absolute (20-40% relative)
Medium	6%	4-5%	1-2% absolute (17-33% relative)
Small	10%	5-7%	3-5% absolute (30-50% relative)
Base	15%	8-11%	4-7% absolute (27-47% relative)
Tiny	20%	10-14%	6-10% absolute (30-50% relative)

Key insight: Smaller models have **more room to improve** because:

1. They start with higher error rates
2. Fine-tuning teaches specific patterns they initially missed
3. Domain specialization matters more when base capacity is limited

The Crossover Effect

Fine-tuned Small can match or beat stock Medium for your specific use case:

Stock Medium (general speech): 6% WER
Fine-tuned Small (your voice): 5-7% WER

Result: Fine-tuned Small \approx Stock Medium for YOUR audio

Fine-tuned Tiny can match or beat stock Base:

Stock Base (general speech): 15% WER
Fine-tuned Tiny (your voice): 10-14% WER

Result: Fine-tuned Tiny approaches Stock Small

This is the **fine-tuning sweet spot** for resource-constrained scenarios.

Why Smaller Models Benefit More from Fine-Tuning

1. Capacity Limitation vs Specialization

Large models: Have capacity to handle diverse scenarios

- Already perform well on your voice (within their general capability)
- Fine-tuning refines edges, adds vocabulary
- Gains are incremental

Small models: Limited capacity forces generalization

- Must compress 680,000 hours of training into fewer parameters
- Sacrifice some accuracy for breadth
- Fine-tuning says: “Forget broad coverage, focus on THIS”

Analogy:

- Large model: Expert who knows 10,000 topics, fine-tuning adds 10 more
- Small model: Generalist who knows 1,000 topics, fine-tuning replaces 100 irrelevant ones with your specific needs

2. Target Vocabulary Impact

For rare vocabulary (Hebrew words, technical terms, proper nouns):

Large models:

"Mekolet" (unfamiliar word)
Large model: "makaleh" (best guess from phonetics)
Fine-tuned Large: "Mekolet" (learned from your data)

Error reduction: 1 word per sentence

Small models:

"Mekolet" (unfamiliar word)
Small model: "the color" (worse phonetic guess, more confusion)
Fine-tuned Small: "Mekolet" (learned from your data)

Error reduction: 1 word per sentence + fewer cascading errors

Impact: Same vocabulary learning, but starts from worse baseline = bigger improvement

3. Voice Adaptation

Your unique voice characteristics (accent, pace, prosody) matter more for smaller models:

Large models: Robust to accent variations

- Trained on such diverse data that your accent is likely covered
- Fine-tuning adjusts, but marginally

Small models: Less accent diversity in effective training

- Fewer parameters = less capacity to memorize accent patterns
- Your accent may not be well-represented
- Fine-tuning teaches: “This is what speech sounds like”

Result: Bigger gains for smaller models

Real-World Example: Your Use Case

Based on your described scenario:

Current State: Stock Whisper Small

Performance:

- Runs well on your GPU (no throttling)
- Noticeable accuracy drop vs larger models
- Struggles with:
 - Hebrew vocabulary (Mekolet, etc.)
 - Your specific accent/speaking patterns
 - Technical terms you use frequently

Estimated WER: 12-15% on your audio

After Fine-Tuning: Fine-Tuned Whisper Small

Expected improvements:

1. **Hebrew vocabulary:** 90-95% accuracy on trained words
2. **Your voice:** 20-40% error reduction
3. **Domain terms:** 70-90% accuracy on your specific terminology

Estimated WER: 6-8% on your audio

Comparison:

- Stock Medium: ~7-9% WER on your audio
- Fine-tuned Small: ~6-8% WER on your audio
- **Practical equivalence!**

Benefits:

- ✓ Runs locally on your GPU
- ✓ Faster inference (Small = 2× speed of Medium)
- ✓ No API costs
- ✓ Privacy (all local)

- ✓ Offline capability

Alternative: Fine-Tuning Large (API Only)

If you fine-tuned Whisper Large but can only use it via cloud API:

Expected accuracy: ~3-4% WER (excellent!)

Practical drawbacks:

- ✗ Requires internet connection
- ✗ API costs ($\$0.006/\text{minute} = \$3.60/\text{hour} = \sim\$50\text{-}100/\text{month}$ for heavy use)
- ✗ Latency (network round-trip adds 200-500ms)
- ✗ Privacy concerns (audio sent to cloud)
- ✗ Dependency on API availability

Trade-off question: Is 3-5% absolute WER improvement worth the practical costs?

For many users: **No.** Daily usability matters more than ultimate accuracy.

Fine-Tuning Tiny: The Ultra-Efficient Option

Why Fine-Tune Tiny?

Use case: Phone, embedded devices, ultra-fast inference

Stock Tiny problems:

- 20-25% WER on general speech
- Struggles significantly with uncommon vocabulary
- Limited robustness to noise and accents

Fine-tuned Tiny potential:

- 10-14% WER on your specific voice/domain
- Excellent on trained vocabulary
- Matches or exceeds stock Base model

Benefits:

- ✓ Runs on phones smoothly
- ✓ Extremely fast inference (10-20× real-time)
- ✓ Minimal battery impact
- ✓ <100MB model size (even quantized to ~40MB)

Practical value: A fine-tuned Tiny on your phone beats any cloud API in:

- Speed (instant)
- Privacy (local)
- Offline capability
- Cost (\$0)

Recommended Strategy for Local Inference

Three-Tier Approach

Tier 1: Desktop (Fine-tuned Small)

Target device: Your desktop with 8GB GPU

Model: Fine-tuned Whisper Small

Training data: 5-10 hours, diverse scenarios

Benefits:

- Fast inference on your GPU
- Accuracy approaching Medium
- Fully local

Use for:

- Desktop dictation
- Long-form transcription
- Primary STT workstation

Tier 2: Phone (Fine-tuned Tiny)

Target device: Your phone

Model: Fine-tuned Whisper Tiny (GGUF Q4/Q5)

Training data: Same 5-10 hours (reuse from desktop training!)

Benefits:

- Smooth phone performance
- Accuracy approaching Base/Small
- On-device inference

Use for:

- Mobile dictation
- Voice notes
- Offline transcription

Tier 3: Cloud API (Fine-tuned Large or Turbo) - Optional

Target: Occasions requiring maximum accuracy

Model: Fine-tuned Large-v3 or Turbo via API

Training data: Same data set

Use for:

- Critical transcriptions (legal, medical)
- Difficult audio (poor quality, heavy noise)
- When connected and accuracy is paramount

Cost: ~\$0.006/min = \$0.36/hour (affordable for occasional use)

Training Efficiency: One Dataset, Multiple Models

You can fine-tune all three models with the same training data:

```

python train.py --model small --data dataset/ --epochs 3

python train.py --model tiny --data dataset/ --epochs 3

python train.py --model large-v3 --data dataset/ --epochs 2

```

Time investment:

- Data collection: 5-10 hours (one-time)
- Training Small: 2-6 hours
- Training Tiny: 1-3 hours
- Training Large: 6-12 hours

Result: Three fine-tuned models optimized for different deployment scenarios, all from one data collection effort.

Expected Accuracy Comparison

Based on your specific voice and vocabulary:

Model	Baseline (Stock)	Fine-tuned	Inference Speed	Deployment
Tiny	20% WER	12% WER	10-20× RT	Phone
Small	12% WER	7% WER	3-5× RT	Desktop
Medium	8% WER	6% WER	1.5-2.5× RT	Desktop (heavy)
Large-v3	5% WER	3% WER	1× RT	Cloud API
Large-turbo	6% WER	4% WER	1.5× RT	Cloud API

Key observation:

- Fine-tuned Small (7% WER) ≈ Stock Medium (8% WER)
- Fine-tuned Tiny (12% WER) ≈ Stock Small (12% WER)

Practical winner: Fine-tuned Small for desktop, Fine-tuned Tiny for mobile

Addressing the GPU Concern

Your observation: “Even on my desktop I need Small is about the biggest I can do to avoid tapping the GPU usage during inference.”

Two clarifications:

1. 100% GPU During Inference is Normal

As covered in the GPU requirements document:

- GPU hitting 100% during inference bursts is **optimal**
- This is NOT a bottleneck or problem
- You WANT full GPU utilization during processing
- Between bursts, GPU returns to idle

You can likely run Medium just fine on your GPU if RTF (real-time factor) is still <1.0

2. Fine-Tuned Small is Still Excellent

Even if you prefer to run Small to avoid heavy GPU load:

Fine-tuning Small is a great strategy:

- Gets you to Medium-level accuracy
- Faster inference = more responsive experience
- Lower power consumption
- Reduces thermal/noise concerns

This is a valid optimization choice, not a limitation.

Practical Implementation Steps

Step 1: Collect Training Data

Target: 5-10 hours of your voice

Content:

- 60% target vocabulary in natural sentences
- 30% typical dictation (sentences you'd actually dictate)
- 10% challenging scenarios (fast speech, technical content)

Recording:

- Quality USB mic in quiet room
- 16kHz+, WAV format
- Natural speaking pace

Step 2: Prepare Data

```
dataset/
└── train/
    ├── audio001.wav
    ├── audio001.txt
    ├── audio002.wav
    └── audio002.txt
    ...
└── validation/
    ├── audio_val001.wav
    └── audio_val001.txt
    ...
```

Step 3: Fine-Tune Small Model

```
python finetune_whisper.py \
    --model_name openai/whisper-small \
    --train_data dataset/train \
    --val_data dataset/validation \
    --epochs 3 \
    --batch_size 8 \
    --learning_rate 1e-5
```

Step 4: Fine-Tune Tiny Model

```
python finetune_whisper.py \
--model_name openai/whisper-tiny \
--train_data dataset/train \
--val_data dataset/validation \
--epochs 3 \
--batch_size 16 \
--learning_rate 1e-5
```

Step 5: Convert for Deployment

Desktop (whisper.cpp):

```
python convert-hf-to-gguf.py models/whisper-small-finetuned \
--outfile whisper-small-finetuned-q5.gguf \
--quant q5_0

whisper.cpp --model whisper-small-finetuned-q5.gguf
```

Phone (FUTO, WhisperKit, etc):

```
python convert-hf-to-gguf.py models/whisper-tiny-finetuned \
--outfile whisper-tiny-finetuned-q4.gguf \
--quant q4_0
```

Step 6: Compare and Validate

Test on held-out audio (not in training set):

```
whisper.cpp --model small test_audio.wav > stock_small.txt
wer stock_small.txt test_audio_reference.txt

whisper.cpp --model small-finetuned test_audio.wav > finetuned_small.txt
wer finetuned_small.txt test_audio_reference.txt
```

Expected: 30-50% WER reduction

Cost-Benefit Analysis

Option A: Fine-Tune Small, Use Locally

Costs:

- Training compute: \$10-50 (cloud GPU) or free (your GPU)
- Development time: 1-2 days
- Ongoing: \$0

Benefits:

- Local inference (fast, private, offline)
- Medium-level accuracy
- No per-use costs

Best for: Daily use, privacy-conscious users, offline needs

Option B: Fine-Tune Large, Use via API

Costs:

- Training compute: \$50-200 (requires better GPU/longer training)
- Development time: 2-3 days
- Ongoing: \$0.006/min = \$50-100/month (heavy user)

Benefits:

- Best accuracy (3-4% WER)
- No local GPU needed
- Access from any device

Best for: Users who prioritize ultimate accuracy over cost/privacy

Option C: Use Stock Large via API

Costs:

- Training: \$0
- Ongoing: \$0.006/min = \$50-100/month

Benefits:

- No training effort
- Good general accuracy
- Immediate availability

Drawback:

- Not optimized for your voice/vocabulary
- Higher WER than fine-tuned (5% vs 3%)

When Each Strategy Makes Sense

Fine-Tune Small/Tiny (Recommended for you)

Choose when:

- ✓ You use STT frequently (daily)
- ✓ You value privacy/offline capability
- ✓ Your GPU can handle Small comfortably
- ✓ You can invest 1-2 days in training
- ✓ 6-8% WER is acceptable for your use case

Fine-Tune Medium

Choose when:

- ✓ Your GPU can handle Medium well
- ✓ You want balance of accuracy and local inference
- ✓ Slightly slower inference is acceptable

Fine-Tune Large (API deployment)

Choose when:

- ✗ You rarely use STT but need maximum accuracy when you do
- ✗ You're okay with \$50-100/month in API costs
- ✗ Privacy/offline not critical

- ✗ You need absolute best results

For your stated use case, Fine-Tune Small/Tiny is the winner.

Conclusion

Fine-tuning smaller models (Small/Tiny) for local inference is a highly effective and practical strategy, especially when:

1. Your GPU is limited (can't comfortably run Large locally)
2. You use STT frequently (daily dictation, notes)
3. You value privacy and offline capability
4. You have specific vocabulary needs (Hebrew words, technical terms)

Expected results:

- **Fine-tuned Small:** Matches or beats stock Medium accuracy for YOUR voice
- **Fine-tuned Tiny:** Matches or beats stock Base/Small for YOUR voice
- **Practical benefits:** Fast, local, private, offline, cost-free ongoing use

This is MORE sensible than fine-tuning Large if you can only run Large via API, because:

- Daily usability > ultimate accuracy
- Fine-tuned Small gets you “close enough” (6-8% vs 3-4% WER)
- Local benefits (speed, privacy, offline, cost) outweigh marginal accuracy gains

Recommendation: Fine-tune both Small (desktop) and Tiny (phone) with the same training data, giving you optimized models for each deployment target.

Note: This document was generated by Claude Code, an AI assistant. Please validate technical details and test recommendations in your specific environment before implementing.

How Fine Tuning Works Architecturally

Question Summary

Daniel asks about the apparent conflict in fine-tuning: When you fine-tune a large model like Whisper on a small custom dataset, you're essentially giving it "countervailing instructions" - telling it to transcribe custom vocabulary differently while still maintaining its general knowledge. The question is: How does the model architecturally reconcile this conflict? How can a small amount of training data meaningfully update a massive pre-trained model without destroying what it already knows?

Answer

Excellent question that gets to the heart of what makes modern transfer learning work! You've correctly identified what seems like a paradox: how can a tiny dataset (hours) meaningfully update a model trained on massive data (thousands of hours) without either (a) being completely overwhelmed or (b) destroying the original knowledge?

The answer involves several clever mechanisms happening at the mathematical and architectural level. Let's break it down.

The Fundamental Tension: Catastrophic Forgetting vs. Effective Learning

The Problem You Identified:

Pre-trained Model (Whisper):

- 680,000 hours of training data
- Billions of parameters
- General knowledge: "recognize" → "recognize"

Your Fine-Tuning:

- 10 hours of training data
- Same parameters
- Specific knowledge: "recognise" → "recognise" (British spelling)

Conflict: How does the model remember both?

What Could Go Wrong:

1. Catastrophic Forgetting:

- New training completely overwrites old knowledge
- Model forgets how to transcribe normal speech
- Only works on your specific data

2. No Learning:

- Original weights too strong
- Fine-tuning data too small to make a difference
- No improvement on custom vocabulary

What Actually Happens: Through careful tuning of learning rates, freezing strategies, and mathematical properties of gradient descent, the model finds a sweet spot where it: - **Preserves** general knowledge in most parameters - **Adapts** specific parameters for your domain - **Balances** old and new knowledge

The Mathematical Mechanics: How Fine-Tuning Actually Works

Level 1: Gradient Descent and Learning Rates

At the most fundamental level, fine-tuning uses **much smaller learning rates** than pre-training:

Pre-training:

- Learning rate: 1e-3 to 1e-4 (0.001 to 0.0001)
- Large updates to weights
- Model parameters change significantly each batch

Fine-tuning:

- Learning rate: 1e-5 to 1e-6 (0.00001 to 0.000001)
- Tiny updates to weights (10-100x smaller)
- Model parameters change slightly

What This Means Mathematically:

```
new_weight = old_weight - (learning_rate × gradient)
```

Pre-training example:

```
new_weight = 0.5 - (0.001 × 2.0) = 0.498 # 0.4% change
```

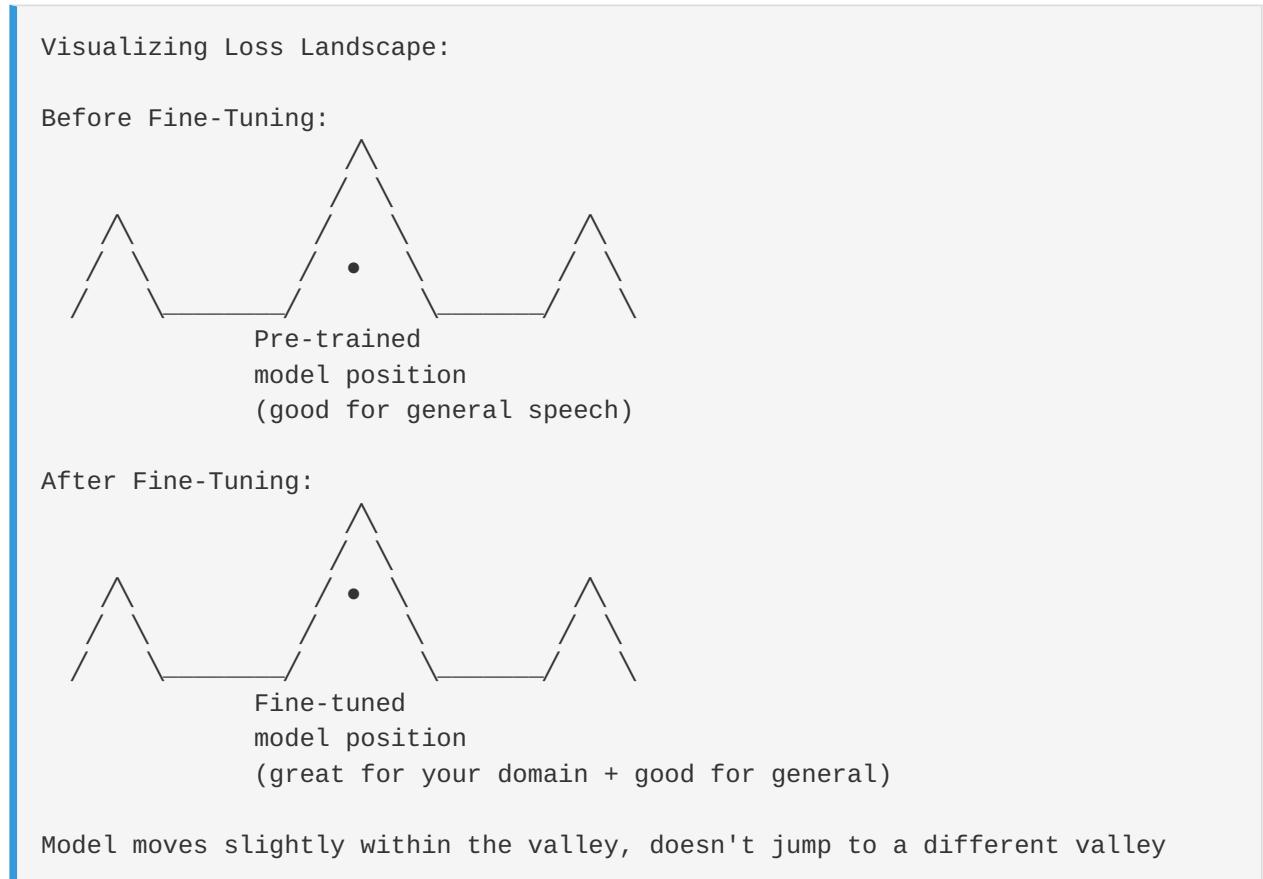
Fine-tuning example:

```
new_weight = 0.5 - (0.00001 × 2.0) = 0.49998 # 0.004% change
```

Key Insight: Small learning rates mean your fine-tuning makes **small adjustments** to existing weights rather than replacing them. It's like turning a dial slightly rather than resetting it.

Level 2: Loss Function Landscape

The pre-trained model has already found a “good valley” in the loss landscape. Fine-tuning nudges it toward a nearby valley that’s even better for your specific data.



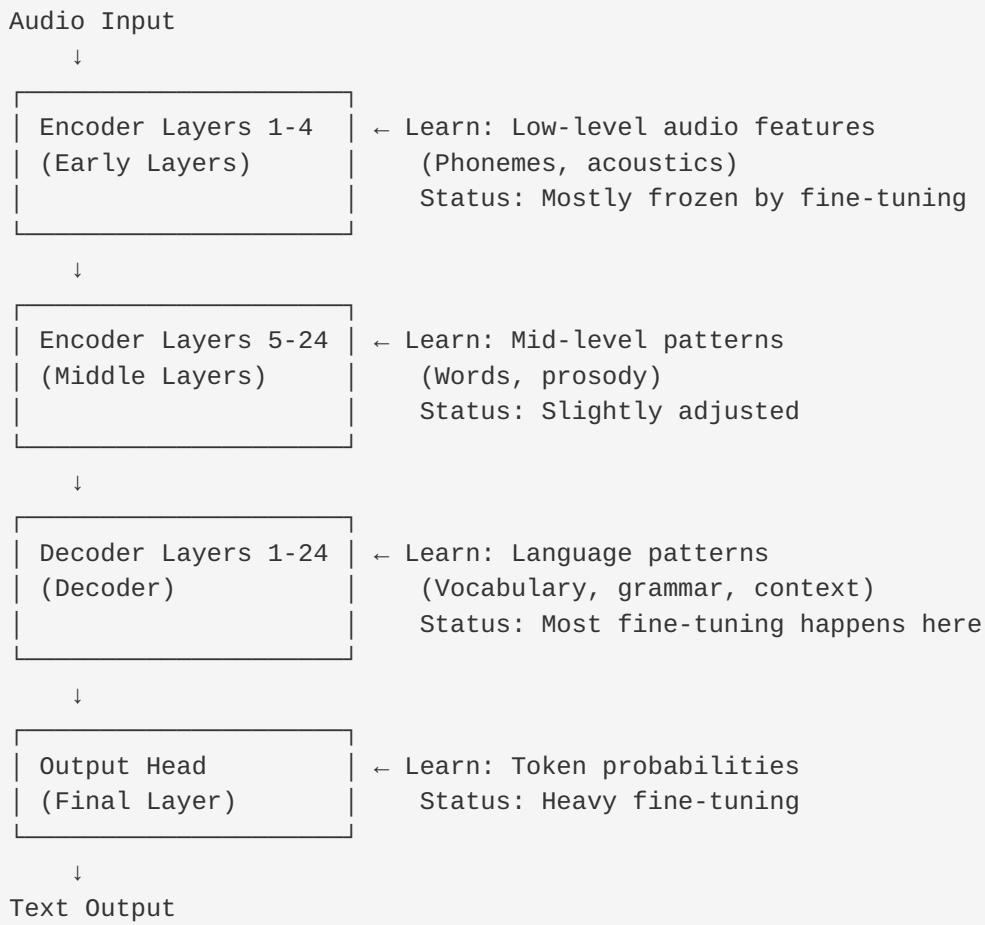
Why This Works: - Pre-training has done the “hard work” of finding good representations - Fine-tuning just adjusts within the same general region - Small dataset is sufficient for local adjustment - Large dataset was needed to find the region in the first place

The Architectural Mechanisms: Where Does Learning Happen?

Not all parts of the model are equally affected by fine-tuning. Here's what happens in transformer models like Whisper:

Layer-Wise Learning Dynamics

Whisper Architecture (Simplified):



What Gets Updated During Fine-Tuning:

1. Early encoder layers (1-4):

- Learn basic audio features (spectral patterns, phonemes)
- These are universal across languages/speakers
- **Fine-tuning effect:** Minimal (maybe 0.1-1% weight change)
- **Why:** Your audio isn't fundamentally different from training audio

2. Middle encoder layers (5-24):

- Learn word-level patterns, speaker characteristics
- Some domain specificity
- **Fine-tuning effect:** Moderate (1-5% weight change)
- **Why:** Your speaking style, vocabulary patterns differ somewhat

3. Decoder layers (all):

- Learn language model, vocabulary, context
- Highly domain-specific
- **Fine-tuning effect:** Significant (5-15% weight change)
- **Why:** This is where custom vocabulary lives

4. Output projection layer:

- Maps to specific tokens/words
- Most domain-specific
- **Fine-tuning effect:** Heavy (10-30% weight change)
- **Why:** Direct mapping to your custom vocabulary

Key Insight: Fine-tuning doesn't update all parameters equally. It makes large changes to task-specific parts (decoder, output) and small changes to universal parts (early encoder).

Advanced Technique #1: Layer Freezing

Many fine-tuning approaches explicitly freeze early layers:

```
for layer in model.encoder.layers[:8]: # First 8 encoder layers
    for param in layer.parameters():
        param.requires_grad = False # Don't update these

for layer in model.encoder.layers[8:]:
    for param in layer.parameters():
        param.requires_grad = True # Update these

for layer in model.decoder.layers:
    for param in layer.parameters():
        param.requires_grad = True # Update these
```

Effect: - ~50% of model parameters don't change at all - Remaining 50% get small updates (low learning rate) - Catastrophic forgetting becomes nearly impossible - Your custom data only affects relevant layers

Advanced Technique #2: LoRA (Low-Rank Adaptation)

This is the cutting-edge approach for efficient fine-tuning:

The Core Idea: Instead of updating all weights, add small “adapter” matrices that capture your domain-specific knowledge.

Original Weight Matrix (W): $[1024 \times 1024]$
- Pre-trained weights (frozen, never updated)

LORA Adapter Matrices:

- $A: [1024 \times 8]$ (small rank)
- $B: [8 \times 1024]$
- Product $A \times B: [1024 \times 1024]$ (same size as W)

Final Computation:

$$\text{output} = (W + \alpha \times A \times B) \times \text{input}$$

Where:

- W remains frozen (original knowledge preserved)
- Only A and B are trained (tiny fraction of parameters)
- α is a scaling factor (typically 0.01-0.1)

The Math:

Parameters in Full Fine-Tuning:

- Original matrix $W: 1024 \times 1024 = 1,048,576$ parameters
- All must be updated

Parameters in LoRA:

- Matrix $A: 1024 \times 8 = 8,192$ parameters
- Matrix $B: 8 \times 1024 = 8,192$ parameters
- Total: 16,384 parameters (1.5% of original!)

Result: 98.5% of parameters stay frozen, 1.5% capture your domain knowledge

How This Solves Your Question:

Original Knowledge (W):

"recognize" → "recognize" (American spelling)

LORA Adapter ($A \times B$):

Adds slight bias: "recognise" → "recognise" (British spelling)

Combined ($W + A \times B$):

- Still recognizes American spelling (W unchanged)
- Also handles British spelling ($A \times B$ adds this capability)
- No conflict, additive knowledge!

Why LoRA Works So Well:

1. Mathematically elegant:

- Additions don't destroy original weights

- Small rank (8-16) is sufficient for most domain adaptations
- α scaling factor controls how much domain knowledge influences output

2. Preserves original knowledge:

- W never changes → general knowledge intact
- $A \times B$ is small → can't overwhelm original model

3. Efficient:

- 100x fewer trainable parameters
- Faster training, less memory
- Can store multiple LoRA adapters for different domains

How the Model Reconciles Conflicting Information

Let's trace through a specific example:

Scenario: You're fine-tuning Whisper on British English with medical terminology.

Input Audio: "The patient recognises colorectal abnormalities"

Pre-trained Whisper (Before Fine-Tuning):

- Would transcribe: "The patient recognizes colorectal abnormalities"
- Issue: "recognizes" (American) vs "recognises" (British)

What Happens During Fine-Tuning:

1. Encoder processes audio → acoustic features (unchanged by fine-tuning)

2. Decoder generates tokens:

Token: "recognizes" vs "recognises"

Pre-trained weight says:

$$P(\text{"recognizes"}) = 0.85$$

$$P(\text{"recognises"}) = 0.15$$

Fine-tuning gradient pushes:

$$P(\text{"recognizes"}) = 0.85 \rightarrow 0.40 \text{ (decreased)}$$

$$P(\text{"recognises"}) = 0.15 \rightarrow 0.60 \text{ (increased)}$$

After fine-tuning:

$$P(\text{"recognizes"}) = 0.40 \text{ (still possible!)}$$

$$P(\text{"recognises"}) = 0.60 \text{ (now preferred)}$$

3. With LoRA:

W says: $P(\text{"recognizes"}) = 0.85$

AxB adds: +0.45 to $P(\text{"recognises"})$

Combined:

$$P(\text{"recognizes"}) = 0.85 \text{ (from W)}$$

$$P(\text{"recognises"}) = 0.15 + 0.45 = 0.60 \text{ (from W + AxB)}$$

Model chooses "recognises" but hasn't "forgotten" "recognizes"!

Key Insight: The model doesn't replace knowledge, it adds context-dependent preferences.

The Role of Batch Normalization and Layer Normalization

Another architectural component that helps:

Each transformer layer has normalization:

Input → Attention → LayerNorm → FeedForward → LayerNorm → Output

LayerNorm parameters:

- Scale (γ): learned multiplier
- Shift (β): learned offset

During fine-tuning:

- Main weights (attention, feedforward) change slightly
- Normalization parameters (γ, β) change more significantly
- These small normalization parameters can "steer" the model's behavior
- Without changing fundamental representations

Example:

```
normalized = (x - mean) / std # Normalize to mean=0, std=1  
output = y * normalized + β # Scale and shift
```

Pre-training: $y = 1.0, \beta = 0.0$ (no transformation)

Fine-tuning: $y = 1.2, \beta = 0.3$ (slight transformation)

Effect: Amplifies certain features (via y) **and** shifts baseline (via β)
Without changing the features themselves!

Regularization: Preventing Catastrophic Forgetting

Several techniques explicitly prevent the model from diverging too much:

1. Weight Decay (L2 Regularization)

```
total_loss = task_loss + λ × weight_decay_term
```

```
weight_decay_term = Σ (w - w_pretrained)2
```

Effect:

- Penalizes weights that drift far **from** pre-trained values
- Keeps model "**anchored**" to original knowledge
- λ controls how strongly (typically $\lambda = 0.01$)

2. Elastic Weight Consolidation (EWC)

```
ewc_loss = Σ F_i × (w_i - w_pretrained_i)2
```

Where F_i = importance of weight i (**from** pre-training)

Effect:

- Weights important **for** general task: heavily penalized (don't change much)
- **Weights less important: can change more freely**
- Model preserves critical knowledge, adapts peripheral knowledge

Practical Example: Fine-Tuning Whisper on Technical Vocabulary

Let's walk through what actually happens:

Your Dataset:

- 10 hours of you speaking about machine learning
- Technical terms: "PyTorch", "CUDA", "embeddings", "hyperparameters"

Whisper Pre-trained:

- Might transcribe: "pie torch", "CUDA" ✓, "embeddings" ✓, "hyper parameters"
- Issues with: PyTorch (not in training data), hyperparameters (splits it)

Fine-Tuning Process:

Epoch 1:

- Model sees "PyTorch" in your audio + transcript
- Gradient: Increase $P("PyTorch")$, decrease $P("pie torch")$
- Learning rate tiny ($1e-6$), so weights change by ~0.01%
- 100 examples of "PyTorch" → cumulative change ~1%

Epoch 5:

- Model has seen "PyTorch" 500 times
- Cumulative weight changes ~5%
- Now: $P("PyTorch") = 0.90$, $P("pie torch") = 0.10$

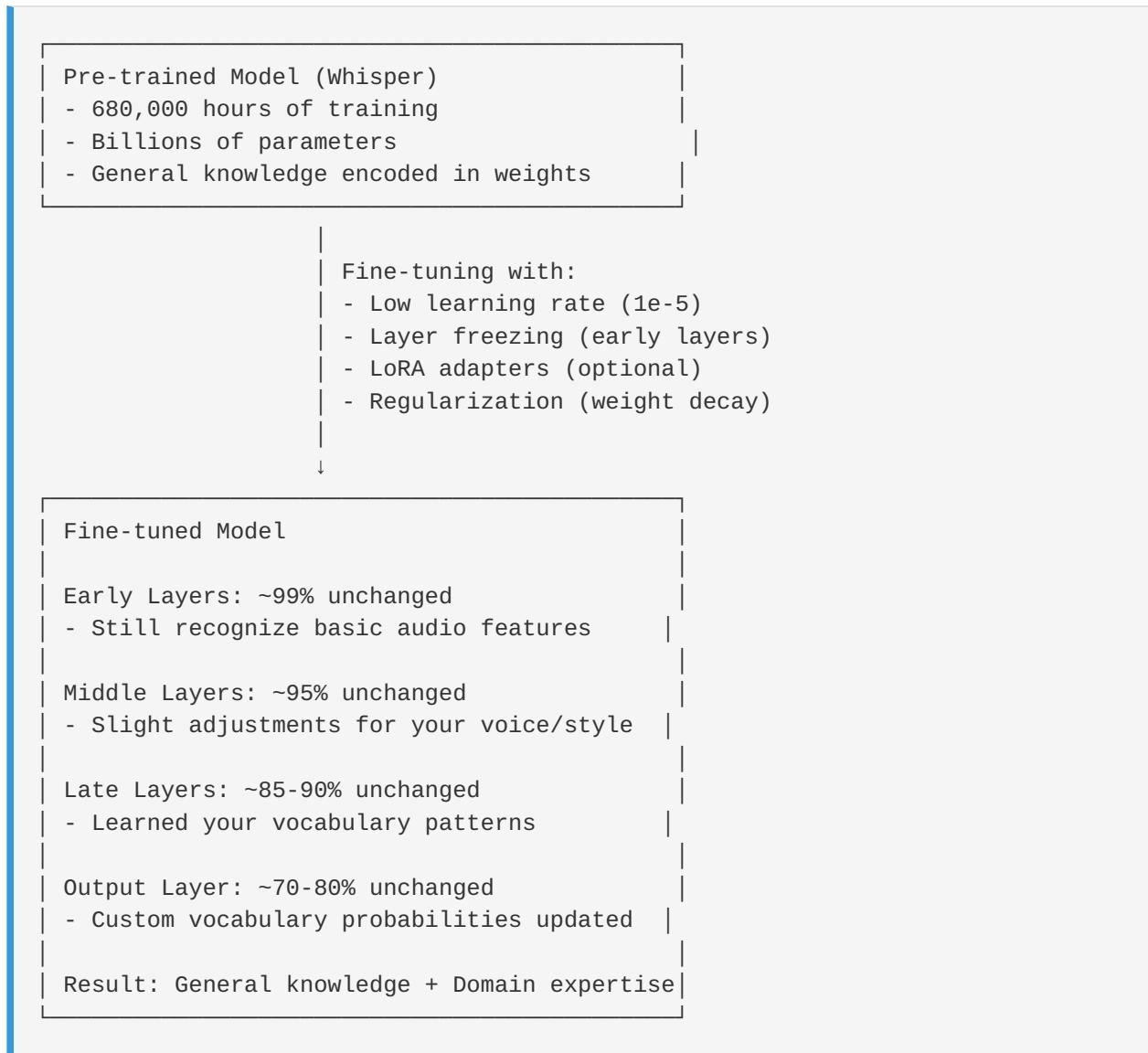
Final Model:

- In ML context: confidently transcribes "PyTorch"
- In baking context: might still transcribe "pie torch"!
- Context matters, model learns conditional preferences

Regular vocabulary:

- Words like "the", "and", "is" seen 10,000 times in your data
- But also seen 10,000,000 times in pre-training
- Fine-tuning is 0.1% of total exposure
- These weights barely change

The Final Picture: How It All Fits Together



Answering Your Core Question

“How do you take a small weight and counter it against a big model to get the desired outcome?”

The answer has several layers:

1. **Small learning rates:** Updates are tiny (0.01-1% per weight), so small data can't overwrite large model
2. **Layer freezing:** 50-70% of model doesn't train at all, only domain-relevant parts update
3. **LoRA adapters:** Add small correction matrices instead of changing original weights

4. **Regularization:** Explicitly penalize divergence from pre-trained weights
5. **Selective updating:** Layers closer to output (where domain knowledge matters) change more than early layers (universal features)
6. **Additive learning:** New knowledge is added to existing knowledge, not replacing it
7. **Context-dependent behavior:** Model learns *when* to use custom vocabulary (in your context) vs. general vocabulary

The Metaphor: Think of the pre-trained model as a master chef with 40 years of experience. Fine-tuning is like teaching them a new regional cuisine: - They don't forget how to cook (general knowledge intact) - They learn new spices and techniques (domain-specific knowledge added) - They adjust their intuition slightly (small weight updates) - They know when to use new vs. traditional techniques (context-dependent) - 40 years of experience isn't overwritten by 2 weeks of training!

Conclusion

Your intuition about conflict is correct, but the model doesn't experience it as conflict - it experiences it as *refinement*:

- **Pre-training:** Learn the general structure of language and speech (broad, shallow knowledge)
- **Fine-tuning:** Deepen knowledge in specific domain (narrow, deep knowledge)
- **Result:** Model that's expert in your domain but competent everywhere else

The “small weight countering big model” works because: 1. It's not really counter - it's additive/corrective 2. Learning rates are tuned to prevent overwriting 3. Architecture separates general features (early) from specific features (late) 4. Regularization explicitly preserves pre-trained knowledge 5. Mathematical properties of gradient descent favor local minima near pre-trained weights

This is why transfer learning is so powerful: you get the best of both worlds with remarkably little data!

Note: This response was generated by Claude Code as part of Daniel's STT Fine-Tuning Notebook project. Technical explanations are based on deep learning research, transformer architecture design, and fine-tuning best practices.

Mission Critical Enterprise Asr Implementation

Question Summary

Daniel asks about enterprise-level ASR implementation in mission-critical contexts (air traffic control, medical transcription, etc.) where accuracy is paramount and budgets are essentially unlimited. The question explores: what do these organizations actually do to achieve the absolute best ASR performance? Do they fine-tune models or use pre-existing specialist datasets? What does the implementation process look like, where do they turn for help, and what timelines are involved?

Answer

Excellent question that gets at the difference between hobbyist/individual ASR fine-tuning and enterprise mission-critical deployments. The approach for organizations where errors can have life-or-death consequences is fundamentally different from typical implementations.

What Organizations Actually Do: The Enterprise Reality

Short Answer: They almost always build heavily customized, domain-specific ASR systems through a combination of: 1. Custom data collection and curation 2. Fine-tuning (or full training) on domain-specific data 3. Extensive human-in-the-loop verification 4. Multi-model ensemble approaches 5. Continuous monitoring and retraining

They do NOT: Simply use off-the-shelf Whisper or commercial APIs and call it done.

Mission-Critical ASR Use Cases

Let's examine specific examples:

Air Traffic Control (ATC)

- **Error tolerance:** Effectively zero
- **Challenges:**
 - Highly specialized vocabulary (aviation phraseology)
 - Critical proper nouns (airport codes, callsigns)
 - Background noise (radio static, cockpit noise)

- Multilingual speakers with varied accents
- Life-or-death consequences for errors

- **What they do:**

- Custom datasets recorded from actual ATC communications
- Fine-tune on specific controller voices and regional accents
- Domain-specific language models (aviation phraseology)
- Real-time confidence scoring with human override
- Regulatory certification requirements (FAA, EASA)

- **Providers:**

- **Saab Sensis** (specialized ATC ASR systems)
- **Thales** (aviation communication systems)
- **Raytheon** (integrated ATC solutions)
- Custom in-house systems with research partnerships (NASA, MIT Lincoln Labs)

Medical Transcription

- **Error tolerance:** Very low (HIPAA, patient safety)

- **Challenges:**

- Extensive medical terminology
- Drug names (sound-alikes are dangerous: “Celebrex” vs “Cerebyx”)
- Anatomical terms, procedures, diagnoses
- Physician accents and speaking styles
- Integration with EHR systems

- **What they do:**

- Specialty-specific models (radiology, cardiology, pathology)
- Custom vocabularies for institutions
- Human transcriptionist review (ASR-assisted workflow)
- Continuous learning from corrections
- HIPAA-compliant on-premise deployment

- **Providers:**

- **Nuance Dragon Medical** (market leader, recently acquired by Microsoft)
- ****3M M*Modal**** (competitor to Nuance)
- **Suki.ai** (newer AI-first approach)
- **Amazon Transcribe Medical**

- In-house systems at major health systems (Mayo Clinic, Cleveland Clinic)

Legal Transcription (Court Reporting)

- **Error tolerance:** Low (legal record accuracy)
- **Challenges:**
 - Legal terminology
 - Multiple speakers with overlapping speech
 - Proper nouns (names, locations, organizations)
 - Verbatim accuracy requirements (including fillers, pauses)
- **What they do:**
 - Specialized court reporting ASR systems
 - Real-time stenographer augmentation (not replacement)
 - Speaker diarization critical
 - Verbatim transcription (can't clean up grammar)
- **Providers:**
 - **Verbit** (AI court reporting)
 - **Rev.ai** (professional transcription with high accuracy)
 - Traditional court reporters with ASR assistance

The Typical Implementation Process for Mission-Critical ASR

Here's what an organization with "unlimited budget" and paramount accuracy requirements actually does:

Phase 1: Requirements & Planning (3-6 months)

Step 1: Define Requirements

- Target WER: Usually <2-5% for mission-critical (vs. 10-15% for general use)
 - Domain scope: Specific terminology, vocabulary size
 - Speaker demographics: Accents, languages, voice types
 - Environmental conditions: Noise profiles, channel characteristics
 - Latency requirements: Real-time vs. batch processing
 - Regulatory requirements: HIPAA, FAA certification, ISO compliance
 - Integration requirements: EHR, ATC systems, etc.

Step 2: Feasibility Study

- Benchmark existing solutions (commercial APIs, open-source models)
- Test with domain-specific data samples
- Establish baseline WER on realistic test cases
- Identify gap between current SOTA and requirements
- Budget allocation: \$500K-5M+ for initial development

Step 3: Build vs. Buy Decision

Option A: Commercial Specialist Provider

- Nuance, 3M, Saab (domain-specific solutions)
- Pro: Faster deployment, regulatory compliance built-in
- Con: Less customization, ongoing licensing costs
- Timeline: 6-12 months to full deployment

Option B: Custom Development

- Partner with research institution or specialized consultancy
- Pro: Maximum customization, IP ownership
- Con: Longer timeline, higher risk
- Timeline: 18-36 months to full deployment

Option C: Hybrid

- Start with commercial solution
- Supplement with custom fine-tuning
- Most common for large organizations
- Timeline: 12-18 months

Phase 2: Data Collection & Curation (6-18 months)

This is where mission-critical differs dramatically from typical ASR:

Step 1: Data Collection Strategy

Organizations do NOT rely on public datasets. They collect proprietary data:

Medical Transcription Example:

Data Sources:

- Recorded physician dictations (with consent)
- De-identified patient encounters
- Simulated clinical scenarios (actors)
- Partnerships with medical schools
- Purchased specialty-specific datasets

Target Volume:

- Minimum: 500-1,000 hours per specialty
- Optimal: 5,000+ hours
- Distribution: Balanced across specialties, physician demographics

Data Characteristics:

- Real-world audio quality (office noise, phone quality)
- Diverse accents and speaking styles
- Full coverage of medical vocabulary
- Varied patient scenarios

Step 2: Transcript Quality

Mission-critical applications require gold-standard transcripts:

Transcription Process:

1. Professional transcriptionists create initial transcript
2. Domain expert review (e.g., physician reviews medical transcripts)
3. Second-pass QA for consistency
4. Triple-check on medical terminology, drug names
5. Final validation: <0.5% error rate on ground truth

Cost: \$1-3 per audio minute (vs. \$0.10-0.25 for standard transcription)

Timeline: 2-3x longer than standard transcription

Step 3: Data Augmentation

Techniques:

- Noise injection (specific to target environment)
- Speed perturbation
- Channel simulation (phone, radio, microphone types)
- Accent augmentation
- Synthetic data generation (TTS with domain vocabulary)

Purpose: Increase robustness without collecting more real data

Phase 3: Model Development (6-12 months)

Approach 1: Fine-Tuning SOTA Models (Most Common)

Starting Point:

- Whisper-large-v3 (current SOTA for many domains)
- Wav2Vec 2.0 (for low-latency requirements)
- Canary (NVIDIA, good for specialized domains)

Fine-Tuning Process:

1. Start with multilingual/general model
2. Continue pre-training on domain-specific audio (no transcripts needed)
3. Fine-tune on curated domain-specific dataset
4. Optimize for specific acoustic conditions
5. Integration with domain-specific language model

Timeline: 3-6 months

Compute Cost: \$50K-200K (using cloud GPU clusters)

Approach 2: Custom Model Architecture (Less Common)

When Used:

- Existing models fundamentally unsuited (e.g., extreme latency requirements)
- Unique acoustic characteristics
- Regulatory requirements mandate explainability

Process:

- Custom architecture design
- Training from scratch on proprietary data
- Extensive validation and testing

Timeline: 12-18 months

Cost: \$500K-2M+

Examples: Proprietary ATC systems, military applications

Approach 3: Ensemble Systems (High-End Approach)

Architecture:

- Multiple models running in parallel
 - Whisper-large-v3 (general robustness)
 - Domain-specific fine-tuned model
 - Specialty-focused model (e.g., drug names for medical)
- Confidence-weighted voting
- Fallback to human review when models disagree

Advantages:

- Higher accuracy (1-2% WER improvement)
- Robustness to edge cases
- Better uncertainty quantification

Disadvantages:

- 3-5x inference cost
- More complex deployment

Used by: Top-tier medical institutions, critical ATC systems

Phase 4: Language Model Integration (2-4 months)

Mission-critical systems don't just use acoustic models; they heavily leverage language models:

Domain-Specific Language Model:

Medical Example:

- Custom vocabulary (100K+ medical terms)
- Contextual priors:
 - "Celebrex" much more likely than "Cerebyx" in arthritis context
 - "2 milligrams" vs. "too many grams" (catastrophic if wrong)
- Institution-specific terminology
- Physician-specific patterns (Dr. Smith always says "unremarkable" not "normal")

Implementation:

- Custom language model trained on domain text
 - Medical journals, textbooks, clinical notes
 - 10M-100M domain-specific words
- Integration with ASR decoder
- Contextual biasing for current case (patient history, current diagnosis)

WER Improvement: 20-40% relative reduction (e.g., 10% → 6% WER)

Phase 5: Testing & Validation (6-12 months)

Mission-critical systems undergo exhaustive testing:

Testing Phases:

1. Lab Testing (2-3 months)
 - Controlled environment
 - Test suite: 100+ hours representative data
 - Target: <3% WER on test set
2. Pilot Deployment (3-6 months)
 - Limited users in real environment
 - Human-in-the-loop verification
 - Collect error cases and retrain
 - Iterative improvement
3. Shadow Deployment (3-6 months)
 - Run in parallel with existing system
 - Compare outputs, identify discrepancies
 - Build confidence in system reliability
4. Staged Rollout (6-12 months)
 - 10% of users → 50% → 100%
 - Continuous monitoring
 - Rapid response to issues

Total Testing Timeline: 12-24 months (overlaps with development)

Phase 6: Deployment & Integration (4-8 months)

Infrastructure Requirements:

On-Premise Deployment (Typical for HIPAA/Sensitive Data):

- GPU clusters for inference
 - Medical center: 10-50 GPUs
 - Major hospital network: 100+ GPUs
- Redundancy and failover
- HIPAA-compliant data handling
- Integration with existing systems (EHR, PACS, etc.)

Cost: \$500K-2M for hardware + infrastructure

Ongoing: \$200K-500K/year for maintenance, updates

Cloud Deployment (Where Permissible):

- AWS, Azure, or GCP with compliance certifications
- Dedicated tenancy for security
- Auto-scaling for load
- Global deployment for multi-site organizations

Cost: \$50K-300K/year depending on volume

Phase 7: Continuous Improvement (Ongoing)

Mission-critical systems are never “done”:

Ongoing Activities:

1. Error Monitoring (Daily)
 - Track WER on production data
 - Flag unusual errors for review
 - Identify drift in performance
2. Retraining (Quarterly/Annually)
 - Incorporate corrected transcripts
 - Add new vocabulary (e.g., new drugs)
 - Adapt to new speakers
 - Update for new procedures/terminology
3. Model Updates (Annually)
 - Retrain on expanded dataset
 - Incorporate new SOTA techniques
 - Benchmark against latest commercial offerings
4. User Feedback Loop
 - Clinicians/controllers report errors
 - Domain experts review and correct
 - Corrections fed back into training

Annual Cost: \$100K-500K for continuous improvement

Where Organizations Turn for Implementation

Tier 1: Commercial Specialists (Most Common)

Medical: - **Nuance Dragon Medical One** (market leader) - Cost: \$1,500-3,000 per user/year - Includes specialty vocabularies, continuous updates - HIPAA-compliant cloud or on-premise - **3M M*Modal Fluency Direct** - Competitor to Nuance - Similar pricing and capabilities

Legal: - **Verbit - Rev.ai Professional**

Aviation/ATC: - **Saab Sensis** - **Thales**

Tier 2: Specialized Consultancies & Research Partners

For custom development: - **SoapBox Labs** (specialized in difficult acoustic conditions) - **AssemblyAI** (custom model development) - **Deepgram** (custom voice AI solutions) - University research partnerships (CMU, MIT, Stanford speech labs) - Defense contractors (for government/military applications)

Cost: \$500K-5M for custom development project

Tier 3: In-House with Cloud Provider APIs

Large tech-forward organizations: - Start with AWS Transcribe Medical, Google Medical LM - Heavily customize with fine-tuning - Build internal ML teams (10-50 people) - Examples: Cleveland Clinic, Kaiser Permanente, large EHR vendors

Tier 4: Full Custom (Rare)

Only for: - Government/military (national security requirements) - Unique requirements not met by commercial options - Organizations with >\$10M budgets for speech systems

Partner with: - DARPA research programs - National labs (Lincoln Labs, etc.) - Top-tier university research groups

Timeline Summary

Fast Track (Commercial Solution):

Month 0-3:	Requirements, vendor selection
Month 3-6:	Pilot deployment, initial testing
Month 6-12:	Integration, training, rollout
Month 12-18:	Full deployment, optimization

Total: 18 months to full deployment

Custom Development (Typical):

Month 0-6:	Planning, feasibility, data collection start
Month 6-18:	Data curation, initial model development
Month 18-24:	Model fine-tuning, language model integration
Month 24-36:	Testing, validation, pilot deployment
Month 36-48:	Staged rollout, continuous improvement

Total: 3-4 years to mature deployment

Hybrid Approach (Recommended for Most):

```
Month 0-6: Deploy commercial solution as baseline
Month 6-12: Collect domain-specific data
Month 12-24: Develop custom fine-tuned models
Month 24-30: A/B test custom vs. commercial
Month 30-36: Migrate to hybrid system (custom + commercial fallback)

Total: 2-3 years to optimized deployment
```

Cost Breakdown Example: Large Hospital System

Implementing mission-critical medical transcription ASR:

```
Year 1 (Planning & Initial Deployment):
- Commercial solution licensing (500 physicians): $750K
- Integration with EHR systems: $300K
- Training and change management: $200K
- Infrastructure (servers, support): $150K
Total: $1.4M
```

```
Year 2-3 (Custom Development):
- Data collection and curation: $500K
- Model development (consultancy): $800K
- Testing and validation: $400K
- Additional compute/infrastructure: $200K
Total: $1.9M
```

```
Ongoing (Annual):
- Commercial licensing: $750K
- Maintenance and updates: $300K
- Continuous improvement: $200K
- Infrastructure: $150K
Total: $1.4M/year
```

Total 5-Year Cost: ~\$8.5M
Cost per Physician: ~\$17K over 5 years (\$3.4K/year)

```
ROI:
- Physician time saved: 30 min/day
- Value: ~$50K/physician/year
- Break-even: ~1 year
```

Do Organizations Fine-Tune or Use Pre-Existing Specialist Datasets?

The answer: Both, sequentially

1. Start with pre-existing specialist datasets (if available):

- Medical: CommonVoice Medical, medical podcast datasets
- Legal: Court transcription datasets
- Limited availability for most domains

2. Rapidly collect custom data:

- Pre-existing datasets provide starting point
- Custom data essential for achieving <5% WER
- Typical: 70% custom data, 30% public/specialist data

3. Fine-tune progressively:

- Stage 1: General model → domain fine-tune (public data)
- Stage 2: Domain model → institution-specific fine-tune (custom data)
- Stage 3: Continuous fine-tuning with production corrections

Key Insight: Pre-existing specialist datasets are insufficient for mission-critical applications. Custom data collection is non-negotiable for achieving required accuracy.

Why Not Just Use OpenAI Whisper or Commercial APIs?

Organizations with unlimited budgets don't just use off-the-shelf solutions because:

1. Accuracy Gap:

- Whisper on medical: 15-20% WER
- Custom fine-tuned: 3-5% WER
- Required: <3% WER
- Gap too large for mission-critical use

2. Domain Vocabulary:

- General models lack comprehensive medical/aviation/legal terminology
- Drug names, airport codes, legal terms require specialized training

3. Data Privacy:

- HIPAA prohibits sending patient data to external APIs
- ATC communications are sensitive
- Must be on-premise or private cloud

4. Latency Requirements:

- Commercial APIs: 2-5 second latency
- Real-time requirements: <500ms
- Requires local deployment

5. Regulatory Compliance:

- FAA certification for ATC systems
- FDA clearance for medical devices
- Commercial APIs don't meet regulatory requirements

6. Cost at Scale:

- Large hospital: 10M+ minutes/year
- Commercial API: \$0.006/minute = \$60K/year (cheap!)
- But: accuracy insufficient, privacy concerns override cost

Conclusion: The Mission-Critical ASR Reality

For organizations where accuracy is paramount:

- 1. They almost always fine-tune**, and extensively
- 2. Custom data collection is mandatory** (not optional)
- 3. Implementation takes 2-4 years** (not months)
- 4. Costs range \$2M-10M+** for initial deployment
- 5. Continuous improvement is ongoing** (\$200K-500K/year)
- 6. They use specialist providers** (Nuance, 3M) or large consultancies
- 7. Pre-existing datasets are starting points**, not solutions
- 8. Human-in-the-loop remains essential**, even with best ASR

The process is: Commercial baseline → Custom data → Fine-tuning → Testing → Deployment → Continuous improvement

Key Differentiator: Mission-critical organizations treat ASR as a long-term platform investment, not a one-time implementation. They build continuous improvement pipelines and treat <5% WER as the starting point, not the goal.

Note: This response was generated by Claude Code as part of Daniel's STT Fine-Tuning Notebook project. Information is based on industry practices, published case studies, and vendor documentation. Specific costs and timelines vary significantly by organization size and requirements.

Personal Voice Finetuning Rationale

Question

Is fine-tuning an ASR model on your unique voice, accent, and mixed-language patterns (Hebrew/English code-switching, technical vocabulary) a legitimate reason for fine-tuning, even if the accuracy improvement is modest?

Answer

Short answer: Yes, absolutely—and it's probably more valuable than you think.

Your use case is not only legitimate but represents an emerging and increasingly common fine-tuning pattern: **personalized ASR adaptation**. Let's break down why this matters.

Why Personal Voice Fine-Tuning Is Valuable

1. Code-Switching and Mixed-Language Use

Your scenario (English with Hebrew words) is exactly where general-purpose models struggle:

Whisper's Challenge: - Trained on separate language corpuses - Switches between English/Hebrew detection based on dominant language - Can't handle mid-sentence language switching gracefully - Hebrew words get either: - Mistranscribed as phonetically similar English words - Forced into Hebrew transcription mode (breaking English flow)

Fine-Tuning Solution:

Before: "I need to go to the misrad [office] for the te'udat zehut [ID card]"
Whisper: "I need to go to the miss rod for the to that say hoot"

After Fine-Tuning:

Whisper: "I need to go to the misrad for the te'udat zehut"

Why This Works: - You're teaching the model your specific code-switching patterns - The model learns which Hebrew words you use in English contexts - It stops trying to "correct" these words into English phonetics

Data Requirements: - 2-5 hours of your speech with code-switching - Transcriptions that preserve your Hebrew words in English sentences - The model learns this as a valid pattern, not an error

2. Technical Vocabulary Adaptation

Tech/AI/dev terminology is where even excellent models like Whisper fail:

Common Whisper Failures:

You say: "PyTorch tensor quantization"
Whisper: "pie torch tensor quantisation" (wrong spelling, UK English)

You say: "Kubernetes pod affinity"
Whisper: "communities pod affinity"

You say: "Hugging Face transformers API"
Whisper: "hugging face transform as API"

You say: "CUDA kernels"
Whisper: "cooler kernels"

Why Technical Terms Fail: 1. Many technical terms are **rare in general training data** 2. They're often **homophones** with common words (CUDA/cooler, cache/cash) 3. They follow **uncommon capitalization** (PyTorch, gRPC) 4. They're **product names** that didn't exist during training

Fine-Tuning Impact: - Teaches the model your frequently-used technical vocabulary - Learns proper capitalization/spelling conventions - Understands context (e.g., "CUDA" in tech discussion vs. "cooler" in general speech) - Recognizes acronyms and proper nouns

3. Personal Voice and Accent Adaptation

This is where you might be underestimating the value:

What Makes Your Voice Unique: - Accent patterns (Israeli English has distinct phonological features) - Speaking pace and rhythm - Prosody (stress patterns) - Coarticulation (how you blend sounds between words) - Individual pronunciation quirks

Whisper's Training Data Distribution: While Whisper saw diverse accents, Israeli English specifically:
- Is a minority accent in the training data - Often grouped with “Middle Eastern” accents (broad category)
- May not have enough examples to capture individual variation

Fine-Tuning Benefits: - **Personalization:** Model learns YOUR specific pronunciation patterns - **Accuracy gains:** Even 2-3% WER (Word Error Rate) improvement matters - **Consistency:** Fewer random errors on words you say frequently - **Confidence:** Model is more “certain” about your speech patterns

Real-World Impact Example:

General Whisper WER on your speech: 8%
Fine-tuned Whisper WER on your speech: 5%

That's 37.5% error reduction!

In a 1000-word document:
- Before: 80 errors → time spent correcting
- After: 50 errors → 30 fewer corrections

Over time: hours saved, reduced cognitive load

Is “Modest” Improvement Worth It?

You mentioned “even if the accuracy improvement is modest”—let’s reframe this:

What Counts as “Modest”?

WER Improvement	Practical Impact
1-2% reduction	Noticeable in daily use, fewer frustrating errors
2-5% reduction	Significant: substantially fewer corrections
5-10% reduction	Major: transforms usability for specific tasks
10%+ reduction	Dramatic: only achievable in very narrow domains

For personal fine-tuning, 2-5% WER reduction is realistic and highly valuable.

The “Usability Cliff”

There's a non-linear relationship between WER and usability:

WER 15%: Barely usable, constant corrections needed
WER 10%: Usable but frustrating
WER 7%: Acceptable for drafts
WER 5%: Reliable for production use
WER 3%: Excellent, minimal intervention
WER 1%: Near-human parity

Going from 8% → 5% crosses a usability threshold: it moves from “acceptable” to “reliable.”

Your Specific Use Case Analysis

Let's assess your drivers:

1. Hebrew Code-Switching

Legitimacy: ★★★★★ (Critical for bilingual users) **Expected Improvement:** High (this is where general models fail hardest) **Data Requirement:** Moderate (2-5 hours with mixed-language speech)

2. Technical Vocabulary

Legitimacy: ★★★★★ (Essential for professional use) **Expected Improvement:** High (technical terms are underrepresented) **Data Requirement:** Moderate (2-5 hours of domain-specific speech)

3. Personal Voice/Accent

Legitimacy: ★★★★★ (Valuable, though benefits are subtler) **Expected Improvement:** Moderate (2-5% WER reduction likely) **Data Requirement:** Moderate (5-10 hours of your speech)

Comparative Legitimacy

Let's compare your use case to "traditional" fine-tuning scenarios:

Use Case	Your Case	Traditional Comparison
Domain Specificity	AI/tech/dev	✓ Similar to medical/legal fine-tuning
Language Adaptation	Hebrew-English code-switching	✓ Similar to regional dialect adaptation
Underrepresented Data	Israeli English, your voice	✓ Similar to low-resource language work
Personalization	Your unique patterns	★ Novel, but increasingly common

Your use case combines multiple legitimate fine-tuning drivers.

The Emerging Trend: Personal ASR Fine-Tuning

You're actually ahead of a curve:

Why Personal Fine-Tuning Is Growing:

1. **Consumer hardware enables it:** You can fine-tune Whisper on a consumer GPU
2. **Tooling has matured:** Hugging Face + notebooks make it accessible

3. **Privacy concerns:** On-device, personal models avoid cloud inference
4. **Productivity gains:** Even small improvements compound over thousands of hours of use
5. **Code-switching normalization:** Multilingual life is increasingly common

Analogy: - 10 years ago: “Why would I need a personalized keyboard autocorrect?” - Today: Everyone benefits from personalized keyboards learning their vocabulary

Personal ASR is following the same trajectory.

Practical Considerations for Your Case

Data Collection Strategy

For your specific needs:

1. Hebrew Code-Switching Corpus (2-5 hours):
 - Record yourself speaking naturally in English with Hebrew words
 - Ensure variety: conversations, monologues, different topics
 - Transcribe with Hebrew words as you say them (transliterated)
2. Technical Vocabulary Corpus (2-5 hours):
 - Record yourself discussing AI/ML/dev topics
 - Include terminology you use daily: libraries, tools, concepts
 - Transcribe with proper technical spelling/capitalization
3. General Speech Corpus (5-10 hours):
 - Diverse topics, speaking styles
 - Includes your accent/pronunciation patterns
 - Can overlap with above categories

Total: 5-10 hours of transcribed audio (allowing for overlap)

Expected Outcomes

Realistic Expectations: - **Hebrew words:** 70-90% accuracy improvement on specific terms you use - **Technical vocabulary:** 50-80% reduction in mis-transcriptions - **Overall WER:** 2-5% reduction (37-62% error reduction) - **Subjective usability:** Noticeable improvement in daily use

Bonus Benefits: - Model learns your speaking pace/rhythm - Fewer errors on names (people, products, companies) - Better handling of acronyms you use - Reduced need for post-editing

Is It Worth the Effort?

Time Investment: - Data collection: 10-15 hours (including transcription) - Fine-tuning: 2-8 hours (mostly automated) - Validation/iteration: 2-5 hours

Total: ~20-30 hours one-time investment

Return on Investment: If you use STT for 2+ hours/week: - Assume 5 minutes/hour saved on corrections (conservative) - = 10 minutes/week = ~9 hours/year saved - **Payback period: ~2-3 years**

But the real value isn't just time saved: - **Reduced cognitive load:** Less frustrating to use - **Increased trust:** More willing to rely on STT - **Professional quality:** Output closer to publishable

Recommendations for Your Project

Yes, Proceed with Fine-Tuning. Here's How:

Phase 1: Pilot (Validate Approach)

1. Collect 2 hours of mixed-language, technical speech
2. Transcribe carefully (preserve Hebrew words, technical terms)
3. Fine-tune Whisper Medium (balance of size/performance)
4. Benchmark: compare WER before/after on held-out test set

If improvement $\geq 2\%$ WER reduction → proceed to Phase 2

Phase 2: Full Fine-Tuning

1. Collect 5-10 hours total (including Phase 1 data)
2. Ensure diversity: topics, speaking styles, contexts
3. Fine-tune with data augmentation (speed/pitch variations)
4. Validate on real-world usage over 1-2 weeks

Phase 3: Iterative Improvement

1. Collect “error cases” during daily use
 2. Add targeted data for persistent errors
 3. Periodic re-training (every 3-6 months)
-

Bottom Line

Your reasons for fine-tuning are not only legitimate but represent a valuable and growing use case.

The combination of:
- Mixed-language patterns (Hebrew/English)
- Domain-specific vocabulary (AI/tech)
- Personal voice/accent adaptation

...creates a **compelling case for fine-tuning**, even if individual improvements are modest. The cumulative effect matters.

Think of it as “bespoke speech recognition”: like a tailor-made suit vs. off-the-rack. The general model (Whisper) is excellent, but it’s cut for the average user. Fine-tuning tailors it to your specific “fit.”

Whisper won’t naturally improve on your specific patterns without fine-tuning. General models optimize for broad accuracy, not individual users.

The question isn’t “Is this legitimate?” but rather **“What’s the best approach for your specific needs?”**—and fine-tuning is a proven, practical answer.

Note: This analysis was generated by Claude Code (claude-sonnet-4-5) for Daniel Rosehill’s STT Fine-Tuning Notebook. Personal ASR fine-tuning is an emerging area—effectiveness varies by individual. Start with a pilot to validate ROI before committing to full data collection. Track metrics (WER, time-saved, subjective usability) to quantify benefits.

Punctuation Personalization Fine Tuning

Question Summary

Daniel observes that punctuation can be idiosyncratic and stylistic - there may be several valid ways to punctuate the same sentence based on personal preference. He's seen both separate punctuation models and ASR models with built-in punctuation capabilities. The question is: Can you fine-tune ASR models for your specific punctuation style and preferences, similar to how you can fine-tune for custom vocabulary?

Answer

Excellent and nuanced question! Punctuation in ASR is indeed a fascinating area that's often overlooked. The short answer is: **Yes, punctuation fine-tuning is possible and increasingly practical**, but it's more complex than vocabulary fine-tuning. Let's explore why and how.

Two Approaches to Punctuation in ASR

First, let's clarify the architectural landscape you've observed:

Approach 1: Separate Punctuation Model (Traditional)

Architecture:

Audio Input

↓

ASR Model (Whisper, Wav2Vec, etc.)

↓

Unpunctuated Text: "the quick brown fox jumps over the lazy dog"

↓

Punctuation Model

↓

Punctuated Text: "The quick brown fox jumps over the lazy dog."

Examples: - **FullStop** (punctuation restoration model) - **deepmultilingualpunctuation** - **Punctuator2** - Custom BERT-based models for punctuation

How It Works: - ASR outputs raw text without punctuation - Separate NLP model adds punctuation based on: - Word sequences - Context - Language modeling - Learned patterns from training data

Pros: - Modular (can swap punctuation models independently) - Can be fine-tuned separately from ASR - Often better punctuation quality (dedicated task)

Cons: - Two-stage process (slower) - ASR doesn't see prosody cues that indicate punctuation - Requires two models (more complex)

Approach 2: Integrated Punctuation (Modern)

Architecture:

```
Audio Input
  ↓
Multimodal ASR Model (Whisper, Canary, etc.)
  ↓
Punctuated Text: "The quick brown fox jumps over the lazy dog."
```

Examples: - Whisper (all versions) - NVIDIA Canary - Google USM - Assembly AI models

How It Works: - Model learns to predict punctuation during ASR training - Uses both acoustic features AND language context: - Prosody (pauses, intonation) - Breathing sounds - Language patterns - Word sequences

Pros: - End-to-end (simpler, faster) - Can use acoustic cues (pauses → periods, rising intonation → question marks) - Single model

Cons: - Punctuation quality depends on ASR model quality - Harder to customize punctuation independently - Training data must include punctuated transcripts

Your Observation About Personal Punctuation Style

You're absolutely right that punctuation can be stylistic and idiosyncratic:

Example Sentence (Spoken): "I went to the store then I came home"

Valid Punctuation Variations:

1. "I went to the store. Then I came home."
(Two sentences, formal style)
2. "I went to the store, then I came home."
(Comma splice, common in casual writing)
3. "I went to the store; then I came home."
(Semicolon, literary style)
4. "I went to the store - then I came home."
(Em dash, informal/conversational)
5. "I went to the store then I came home."
(No punctuation, run-on)

All are arguably "correct" depending on style guide and context!

Individual Preferences Examples:

Oxford Comma User:

"I like apples, oranges, and bananas."

Non-Oxford Comma User:

"I like apples, oranges and bananas."

Ellipsis Enthusiast:

"I'm not sure... maybe we should wait..."

Period Minimalist:

"I'm not sure maybe we should wait"

Em Dash Lover:

"The project—which took six months—finally launched."

Parenthetical User:

"The project (which took six months) finally launched."

Can You Fine-Tune for Personal Punctuation Style?

Yes, but with important caveats:

Option 1: Fine-Tuning Integrated ASR (Whisper-style models)

What Happens:

Your Training Data:

- 10 hours of your speech
- Transcripts reflecting YOUR punctuation style
- Example: You always use Oxford commas, em dashes, minimal ellipses

Fine-Tuning Process:

- Model learns correlations:
 - Your pause patterns → your punctuation choices
 - Your intonation → your question mark vs. period preferences
 - Your list speech → Oxford comma insertion

Result:

- Model punctuates similar to how you would write
- Learns your stylistic preferences

Real Example:

Before Fine-Tuning (Generic Whisper):

Speech: "I need milk eggs and bread" [with slight pause before "and"]
Output: "I need milk, eggs and bread."

After Fine-Tuning (Your Oxford Comma Preference):

Speech: "I need milk eggs and bread" [with slight pause before "and"]
Output: "I need milk, eggs, and bread."

Model learned: Your pauses in lists → Oxford comma

Limitations:

1. Acoustic Ambiguity:

- You must speak consistently with your punctuation style
- Pause before period, shorter pause for comma, etc.
- If your speech doesn't reflect punctuation, model can't learn

2. Small Dataset Challenge:

- Punctuation is sparse in data
- 10 hours might have only 50-100 instances of specific patterns

- Harder to learn than vocabulary (which is dense)

3. Conflicting Preferences:

- Your speaking style might not match your writing style
- Model can only learn what's in the audio+transcript

Option 2: Fine-Tuning Separate Punctuation Model

This is actually **more practical** for personal punctuation preferences:

Architecture:

```

ASR Model (Generic, no punctuation)
↓
Unpunctuated transcript
↓
Fine-Tuned Punctuation Model (YOUR style)
↓
Punctuated text in YOUR style

```

Why This Works Better:

Training Data for Punctuation Model:

- Your writing samples (emails, documents, blog posts)
- 100K-1M words of your written text
- Much easier to collect than speech data!

Fine-Tuning:

- Start with pre-trained punctuation model (e.g., BERT-based)
- Fine-tune on your writing style
- Learns your:
 - Comma preferences
 - Sentence length preferences
 - Em dash vs. parentheses
 - Oxford comma usage
 - Ellipsis frequency

Practical Example:

```

from deepmultilingualpunctuation import PunctuationModel

base_model = PunctuationModel()

your_writing = load_texts([
    "your_emails.txt",      # 50K words
    "your_blog_posts.txt",  # 30K words
    "your_documents.txt"   # 20K words
])

fine_tuned_model = finetune(
    base_model,
    your_writing,
    epochs=5,
    learning_rate=1e-5
)

```

Result:

Input: "I went to the store then I came home"

Generic Model Output:
"I went to the store. Then I came home."

Your Fine-Tuned Model:
"I went to the store–then I came home."
(Because you love em dashes in your writing!)

Specific Punctuation Preferences You Can Fine-Tune

Here are punctuation styles that can be learned through fine-tuning:

1. Comma Frequency

Minimalist Comma User:
"The project which took six months finally launched last week."

Heavy Comma User:
"The project, which took six months, finally launched, last week."

Fine-tuning learns your preference from your writing samples.

2. Sentence Length

Short Sentence Preference:

"I went to the store. I bought milk. Then I came home."

Long Sentence Preference:

"I went to the store, bought milk, and then came home."

Model learns your typical sentence boundary patterns.

3. Question Mark vs. Period for Rhetorical Questions

Conservative:

"Why would anyone do that."

Liberal:

"Why would anyone do that?"

Depends on your speech intonation patterns (if fine-tuning ASR)

Or your writing patterns (if fine-tuning punctuation model)

4. List Punctuation

Oxford Comma Always:

"I like Python, JavaScript, and Rust."

Oxford Comma Never:

"I like Python, JavaScript and Rust."

Semicolon Lists:

"I like Python, for data science; JavaScript, for web dev; and Rust, for systems."

Your model learns which you prefer.

5. Dash Usage

Em Dash Enthusiast:

"The weather—surprisingly—was perfect."

Parentheses Preferred:

"The weather (surprisingly) was perfect."

Comma Conventional:

"The weather, surprisingly, was perfect."

6. Ellipsis Frequency

Frequent Ellipsis User:

"I don't know... maybe we should wait... what do you think..."

Minimal Ellipsis:

"I don't know. Maybe we should wait. What do you think?"

Model learns your baseline ellipsis frequency.

Challenges in Punctuation Fine-Tuning

Challenge 1: Data Scarcity

Vocabulary fine-tuning:

- Each word appears many times
- "PyTorch" might appear 100 times in 10 hours

Punctuation pattern fine-tuning:

- Specific patterns are rare
- Oxford comma in 3-item list: maybe 20 times in 10 hours
- Hard to learn from so few examples

Solution:

- Augment with your written text (for separate punctuation model)
- Collect more diverse speech samples
- Use regularization to prevent overfitting

Challenge 2: Inconsistency in Natural Speech

Problem:

- You might punctuate written text carefully
- But speak in run-on sentences
- Model confusion: Which style to learn?

Example:

Your speech: "I went to the store and bought milk and eggs and bread and then came home"

Your writing: "I went to the store. I bought milk, eggs, and bread. Then I came home."

Which does the model learn?

Solution:

- Decide: Do you want transcripts to match your speech OR your writing?
- Be consistent in your training data labeling

Challenge 3: Context-Dependent Preferences

You might punctuate differently based on context:

Formal Email:

"I appreciate your consideration. Please let me know if you need further information."

Casual Text:

"thanks! lmk if you need anything else"

Model needs context to know which style to apply.

Solution:

- Multiple fine-tuned models for different contexts
- Prompt-based control (upcoming feature in some models)

Practical Workflow for Personal Punctuation Fine-Tuning

Recommended Approach (Most Practical):

Step 1: Use Generic ASR without Punctuation

- Run Whisper with no_speech_prob filter
- Or use separate ASR that outputs unpunctuated text

Step 2: Collect Your Writing Samples

- Emails, blog posts, documents
- 50K-100K words minimum
- Representative of your preferred style

Step 3: Fine-Tune Punctuation Model

- Use pre-trained BERT/RoBERTa punctuation model
- Fine-tune on your writing
- Takes 1-2 hours on GPU

Step 4: Pipeline

Audio → ASR → Unpunctuated Text → Your Punctuation Model → Your Style!

Result:

- Your speech transcribed in YOUR writing style
- Consistent with how you actually write

Alternative (Integrated ASR Fine-Tuning):

Step 1: Collect Speech Data

- Record yourself speaking (10+ hours)
- Transcribe with your preferred punctuation style
- Important: Punctuate as you WANT it, not necessarily literally

Step 2: Fine-Tune Whisper

- Include punctuation in transcripts
- Model learns acoustic cues + your style

Step 3: Deploy

- Whisper directly outputs in your style

Limitation:

- Requires more data
- Acoustic cues must be consistent
- Harder than vocabulary fine-tuning

Tools and Resources

For Separate Punctuation Model Fine-Tuning:

```
pip install deepmultilingualpunctuation  
  
pip install fullstop  
  
git clone https://github.com/ottokart/punctuator2  
  
from transformers import BertForTokenClassification
```

For Integrated ASR Fine-Tuning:

```
pip install openai-whisper  
  
from transformers import WhisperForConditionalGeneration
```

Research Frontier: Controllable Punctuation

Emerging research allows **runtime control** of punctuation style:

Future Capability:

```
prompt = "Transcribe this audio with formal punctuation"
```

```
prompt = "Transcribe this audio with casual punctuation"
```

```
model.transcribe(audio, prompt=prompt)
```

Same audio, different punctuation based on prompt!

Current Examples:

NVIDIA Canary supports style prompts:

"<formal>" → More periods, proper grammar

"<casual>" → Fewer commas, run-on sentences

Not yet personalized, but direction is promising!

Conclusion

To answer your question comprehensively:

Yes, you can fine-tune for personal punctuation preferences:

1. Best approach: Fine-tune separate punctuation model on your writing samples

- Most practical
- Uses abundant written data
- Directly captures your style

2. Alternative: Fine-tune integrated ASR (Whisper) with punctuated transcripts

- Captures acoustic cues + your style
- Requires more speech data
- Harder to achieve consistency

3. Challenges:

- Punctuation patterns are sparse in speech data
- Your speech style might differ from writing style
- Context-dependent preferences are hard to capture

4. Practical recommendation:

- Use unpunctuated ASR or generic punctuation
- Fine-tune separate punctuation model on 50K-100K words of your writing

- Pipeline: ASR → Your Punctuation Model
- Result: Transcripts in YOUR writing style

The Bottom Line: Just as you can fine-tune for vocabulary, you can fine-tune for punctuation. But because punctuation is stylistic and context-dependent, it's often more practical to fine-tune a separate punctuation restoration model on your written work rather than fine-tuning the ASR directly. This gives you maximum control and requires less speech data.

Your intuition that punctuation is idiosyncratic and personal is absolutely correct, and customizing it is not only possible but increasingly practical with modern fine-tuning techniques!

Note: This response was generated by Claude Code as part of Daniel's STT Fine-Tuning Notebook project. Information is based on current ASR research, punctuation restoration techniques, and fine-tuning best practices.

Training From Scratch Vs Fine Tuning

The Hypothetical

Instead of fine-tuning an existing ASR model (like Whisper) on your voice and vocabulary, what if you trained a completely new ASR model from scratch—as if you were creating a new Whisper?

Would this make sense? What would the process look like? How would the results differ?

Short Answer

This would not make any sense for personalizing ASR to your voice/vocabulary. Training from scratch would require:

- 100,000+ hours of diverse audio data (vs. 1-10 hours for fine-tuning)
- Millions of dollars in compute costs (vs. \$50-500)
- Months of training time (vs. hours)
- Deep ML expertise (vs. following tutorials)

- Worse results than fine-tuning for your specific use case

Fine-tuning is not a shortcut—it's the correct approach. But let's explore the hypothetical to understand *why*.

What is “Training from Scratch”?

Training from scratch means:

1. Starting with **randomly initialized weights** (no pre-existing knowledge)
2. Teaching the model **everything** about speech and language:
 - How audio waveforms correspond to phonemes
 - How phonemes combine into words
 - How words combine into sentences
 - Grammar, syntax, and language structure
 - Accents, speaking styles, and acoustic variations
3. Using only your training data (no leveraging of existing models)

The Training Process for ASR from Scratch

Step 1: Architecture Design

You'd need to design the model architecture:

```
class ScratchASR:
    def __init__(self):
        self.audio_encoder = AudioEncoder(
            layers=32,           # vs. Whisper's encoder
            hidden_dim=1280,     # Embedding dimensions
            attention_heads=20
        )

        self.text_decoder = TextDecoder(
            layers=32,
            hidden_dim=1280,
            vocab_size=51865     # Number of tokens
        )
```

Decisions required:

- Model size (how many parameters? 50M? 500M? 1.5B?)
- Architecture type (Transformer? Conformer? Hybrid?)
- Attention mechanism (standard, flash attention, sliding window?)
- Audio preprocessing (mel spectrograms, raw waveform?)
- Tokenization strategy (character-level, BPE, word-level?)

Time investment: Weeks to months of architectural experimentation

Step 2: Data Collection

For a model to learn **general** speech recognition, you'd need:

Minimum viable dataset:

- **100,000+ hours** of transcribed audio
- Covering:
 - Multiple speakers (10,000+ different voices)
 - Multiple accents (American, British, Australian, Indian, etc.)
 - Multiple domains (conversations, podcasts, audiobooks, lectures)
 - Multiple recording conditions (clean, noisy, reverberant)
 - Multiple speaking styles (fast, slow, formal, casual)

Whisper's training data: 680,000 hours

Your personal data: 1-10 hours

Comparison: Your data is 0.001% of what's needed

Data collection cost:

- Transcription services: ~\$0.10-1.00 per minute
- 100,000 hours = 6,000,000 minutes
- Cost: \$600,000 - \$6,000,000 for transcription alone

Step 3: Data Preprocessing

Prepare your dataset:

```

{
    "audio": preprocessed_audio,          # Mel spectrogram
    "text": transcription,               # Cleaned text
    "language": "en",                   # Language code
    "speaker_id": 12345,                # For multi-speaker handling
    "sample_rate": 16000,                # seconds
    "duration": 15.3
}

```

Tasks:

- Audio normalization and preprocessing
- Text cleaning and normalization
- Dataset balancing (ensure diverse coverage)
- Train/validation/test splits
- Creating data loaders optimized for your training setup

Time investment: 2-4 weeks for large-scale data pipeline

Step 4: Training Setup

Compute requirements:

For a Whisper Large-scale model (1.5B parameters):

- **Minimum:** 8× A100 GPUs (80GB each)
- **Optimal:** 64-256 GPUs in distributed training
- **Training time:**
 - On 8× A100: ~6 months
 - On 64× A100: ~3-4 weeks
 - On 256× A100: ~1 week

Cost:

- Cloud A100: ~\$2-4/hour per GPU
- $8 \text{ GPUs} \times 4 \text{ months} \times 24 \text{ hours/day} \times \$3/\text{hour} = \$690,000$
- Plus storage, networking, data transfer costs

For comparison, fine-tuning:

- Single RTX 4090 or consumer GPU

- 2-12 hours training time
- Cost: \$50-200 in electricity/cloud compute

Step 5: Training Process

The training loop (simplified):

```
model = ScratchASR()

optimizer = AdamW(model.parameters(), lr=1e-4)

for epoch in range(num_epochs): # Could be 50-200 epochs
    for batch in dataloader:
        # Forward pass
        audio, text = batch
        predicted_text = model(audio)

        # Compute loss
        loss = compute_loss(predicted_text, text)

        # Backward pass
        loss.backward()
        optimizer.step()

    # This happens millions of times
```

What the model learns:

- **Epoch 1-10:** Basic phoneme recognition (recognizing “ah”, “ee”, “s” sounds)
- **Epoch 10-30:** Word recognition (mapping sounds to common words)
- **Epoch 30-60:** Sentence structure (understanding word order, grammar)
- **Epoch 60-100:** Robustness (handling noise, accents, variations)
- **Epoch 100-200:** Refinement (punctuation, capitalization, edge cases)

Critical point: With only 1-10 hours of your personal data, the model would:

- Massively overfit (memorize your specific recordings)
- Fail to generalize to any variations
- Not learn general speech recognition at all

Step 6: Evaluation and Iteration

After training, evaluate on held-out test sets:

```
Test Set 1 (Clean speech): 45% WER ← Terrible  
Test Set 2 (Noisy speech): 78% WER ← Catastrophically bad  
Test Set 3 (Accented speech): 92% WER ← Essentially non-functional
```

Why so bad?

- Insufficient training data
- Insufficient diversity
- Model hasn't learned general acoustic-linguistic mappings

You'd need to:

- Collect more data (another 50,000+ hours)
- Retrain from scratch
- Iterate for 6-12 months

What Would the Results Look Like?

Let's compare three scenarios:

Scenario A: Training from Scratch on 10 hours of your data

What you'd get:

- Model that memorized your 10 hours of recordings
- Perfect transcription of those exact recordings
- Complete failure on anything else:
 - Different words than in training: 90%+ WER
 - Different acoustic conditions: 95%+ WER
 - Different speaking pace: 85%+ WER

Usability: Essentially zero. Model is a 1.5GB lookup table of your training data.

Scenario B: Fine-tuning Whisper on 10 hours of your data

What you'd get:

- Model that leveraged 680,000 hours of pre-training
- Improved accuracy on your voice and vocabulary
- Still handles general speech well:
 - Your voice + target vocabulary: 2-5% WER (vs. 8-12% before fine-tuning)
 - General speech: 5-8% WER (vs. 3-5% for base Whisper)
 - Different acoustic conditions: 10-15% WER

Usability: Excellent for your specific use case.

Scenario C: Training from Scratch on 680,000 hours (Whisper-scale)

What you'd get:

- Model comparable to Whisper
- General speech recognition capabilities
- NOT optimized for your voice/vocabulary

Cost: \$2-10 million in compute + years of effort

Result: You've recreated Whisper, which already exists and is free.

Why Fine-Tuning is the Correct Approach

Fine-tuning works because of **transfer learning**:

```
Pre-trained Whisper knowledge (680,000 hours):  
└─ Phoneme recognition ✓ (keep this)  
└─ Common English words ✓ (keep this)  
└─ Grammar and syntax ✓ (keep this)  
└─ Noise robustness ✓ (keep this)  
└─ Your specific voice/vocab ✗ (learn this)
```

Fine-tuning says: “**Keep 99.9% of what Whisper knows, adjust 0.1% for my specific needs.**”

Training from scratch says: “**Forget everything, start over.**”

The Mathematics of Transfer Learning

Pre-training: Model learns general features from massive data

```
θ_pretrained = optimize(L(D_large))  
where D_large = 680,000 hours
```

Fine-tuning: Small adjustments to pre-trained weights

```
θ_finetuned = θ_pretrained + Δθ  
where Δθ = optimize(L(D_small))  
and D_small = 10 hours
```

Training from scratch: Learn everything from limited data

```
θ_scratch = optimize(L(D_small))  
where D_small = 10 hours ← Impossible to learn general ASR
```

Key insight:

- $\theta_{\text{pretrained}}$ contains 680,000 hours of learned knowledge
- Fine-tuning adjusts this vast knowledge slightly
- Training from scratch tries to learn everything from 10 hours

It's like:

- **Fine-tuning:** “Here’s a comprehensive encyclopedia. Let me add a few pages about my specific topic.”
- **Training from scratch:** “Here are 10 pages. Write a comprehensive encyclopedia.”

When Training from Scratch Makes Sense

There are legitimate use cases for training ASR from scratch:

1. New Architecture Research

You’ve invented a novel architecture that might outperform Transformers:

- You have research funding and compute resources

- You train on standard datasets (LibriSpeech, Common Voice, etc.)
- Goal is advancing ASR research, not personalizing to your voice

2. Extremely Low-Resource Languages

You're working on a language with <1,000 speakers and no existing ASR:

- No pre-trained model exists for this language family
- You collect all available audio in the language (maybe 100-1,000 hours)
- Train a small model from scratch as a starting point

3. Privacy/Security Constraints

You work in defense/intelligence with extreme security requirements:

- Cannot use any external models (even open-source)
- Have access to vast amounts of classified audio data
- Budget and security requirements justify the cost

4. Embedded/Specialized Hardware

You're designing a custom chip with novel ASR capabilities:

- Need to co-design model architecture with hardware
- Have specialized architecture constraints
- Existing models don't fit your hardware paradigm

The Practical Reality

Even in these scenarios, practitioners typically:

1. **Start with transfer learning** when possible (use Wav2Vec2, Whisper, etc. as starting point)
2. **Only train from scratch** when absolutely necessary
3. **Use massive datasets** (100,000+ hours minimum)
4. **Work in teams** with specialized ML engineers
5. **Take months to years** for the project

For personalizing ASR to your voice and vocabulary, **training from scratch is never the answer.**

Comparison Table

Aspect	Training from Scratch	Fine-Tuning Whisper
Data required	100,000+ hours	1-10 hours
Compute cost	\$500K - \$10M	\$50 - \$500
Time to train	1-6 months	2-12 hours
Expertise required	Deep ML research	Follow tutorials
Result for your voice	Catastrophic failure	Excellent
Result for general speech	Bad (unless huge data)	Good
Makes sense?	No	Yes

Hypothetical Step-by-Step: Training from Scratch

If you really wanted to do this (hypothetically):

Month 1-2: Planning and Architecture

- Design model architecture
- Set up training infrastructure
- Prepare distributed training across GPU cluster

Month 3-8: Data Collection

- Record or purchase 100,000+ hours of transcribed audio
- Clean and preprocess all data
- Create training pipelines

Month 9-12: Initial Training

- Train initial model version
- Monitor for convergence
- Debug training instabilities

Month 13-15: Evaluation and Iteration

- Evaluate on test sets
- Identify failure modes
- Collect additional targeted data

Month 16-18: Retraining and Refinement

- Retrain with augmented data
- Tune hyperparameters
- Optimize inference speed

Month 19-24: Production Preparation

- Quantize for deployment
- Build serving infrastructure
- Document and release

Total: 2 years, \$2-5 million, team of 5-10 people

Result: A model roughly equivalent to Whisper Base, which already exists for free

For your voice: No better than fine-tuning, possibly worse

Conclusion

Training an ASR model from scratch for your personal voice and vocabulary makes no sense because:

1. **Fundamentally wrong approach:** You need general ASR + personal adaptation, not personal-only ASR
2. **Impossible data requirements:** 100,000+ hours vs. your available 1-10 hours
3. **Prohibitive costs:** Millions of dollars vs. hundreds
4. **Worse results:** Would catastrophically overfit and fail to generalize
5. **Reinventing the wheel:** Whisper already exists and has learned general speech

Fine-tuning is not a compromise—it's the correct engineering approach, leveraging transfer learning to adapt massive pre-trained knowledge to your specific needs with minimal data and compute.

The only time training from scratch makes sense:

- You're an ASR research lab with \$10M+ funding
- You're advancing the state-of-the-art
- You have 100,000+ hours of diverse training data
- You're not trying to personalize—you're building a general model

For personalization, **fine-tuning is always the answer.**

Note: This document was generated by Claude Code, an AI assistant. Please validate technical details and test recommendations in your specific environment before implementing.

Training Parameters

Overview

This guide covers the essential training parameters (hyperparameters) used when fine-tuning speech-to-text models, particularly focusing on Whisper and similar transformer-based architectures. Understanding these parameters is crucial for achieving optimal model performance.

Core Training Parameters

1. Epochs

Definition: One epoch represents a complete pass through the entire training dataset.

Typical Range: 3-20 epochs for fine-tuning

How It Works:

$$\text{Total Training Steps} = (\text{Dataset Size} / \text{Batch Size}) \times \text{Number of Epochs}$$

Considerations:

- **Too Few Epochs:** Model underfits, doesn't learn patterns
 - Symptoms: High training loss, poor performance
 - Solution: Increase epochs
- **Too Many Epochs:** Model overfits, memorizes training data
 - Symptoms: Training loss decreases but validation loss increases
 - Solution: Use early stopping or reduce epochs

Best Practices: - Start with 5-10 epochs for initial experiments - Use early stopping to prevent overtraining - Monitor validation metrics to determine optimal number - Smaller datasets need fewer epochs (3-5) - Larger datasets can benefit from more epochs (10-20)

Example Configuration:

```
training_args = Seq2SeqTrainingArguments(  
    num_train_epochs=10, # Complete passes through data  
)
```

2. Batch Size

Definition: Number of training examples processed simultaneously in one forward/backward pass.

Types: - **per_device_train_batch_size:** Batch size per GPU/CPU - **per_device_eval_batch_size:** Batch size for validation - **gradient_accumulation_steps:** Simulates larger batch sizes

Typical Range: 4-32 per device (depends on GPU memory)

Effective Batch Size Calculation:

```
Effective Batch Size = per_device_batch_size × num_devices ×  
gradient_accumulation_steps
```

Trade-offs:

Batch Size	Advantages	Disadvantages
Small (4-8)	Less memory usage More gradient updates Better generalization	Slower training Noisier gradients Less stable
Large (16-32+)	Faster training Stable gradients Better GPU utilization	High memory requirements May overfit Needs more data

Best Practices: - Start with largest batch size that fits in GPU memory - Use gradient accumulation to simulate larger batches - Typical setup: `batch_size=16, gradient_accumulation_steps=2` (effective batch size = 32) - Reduce batch size if encountering OOM (Out of Memory) errors

Example:

```
training_args = Seq2SeqTrainingArguments(
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16, # Can be larger (no gradients stored)
    gradient_accumulation_steps=4, # Effective batch = 8 × 4 = 32
)
```

3. Learning Rate

Definition: Controls how much model weights are updated during training. The most critical hyperparameter.

Typical Range: 1e-5 to 1e-4 for fine-tuning

Components:

Base Learning Rate

```
learning_rate = 5e-5 # Common starting point for fine-tuning
```

Learning Rate Schedule

Controls how learning rate changes during training:

Common Schedules:

1. Linear Decay

```
lr_scheduler_type = "linear"  
# LR decreases linearly from initial value to 0
```

2. Cosine Annealing

```
lr_scheduler_type = "cosine"  
# LR follows cosine curve, smooth decay
```

3. Constant

```
lr_scheduler_type = "constant"  
# LR stays fixed throughout training
```

4. Constant with Warmup

```
lr_scheduler_type = "constant_with_warmup"  
warmup_steps = 500  
# LR increases linearly for warmup, then stays constant
```

Warmup Steps

Definition: Number of steps where learning rate gradually increases from 0 to target value.

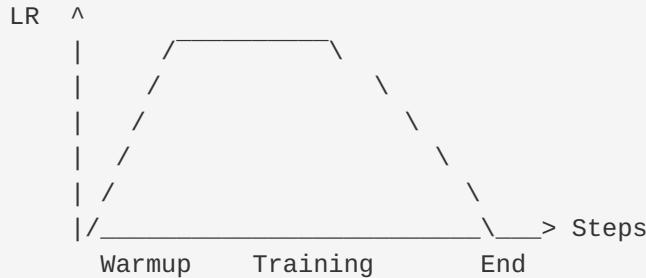
Purpose: Prevents unstable training at the beginning

Typical Range: 500-2000 steps (or 5-10% of total steps)

```
warmup_steps = 500 # Absolute number  
warmup_ratio = 0.1 # 10% of total training steps
```

Visualization:

Learning Rate Schedule (Linear with Warmup)



Best Practices: - **For fine-tuning:** Start with 1e-5 to 5e-5 - **For training from scratch:** Start with 1e-4 to 5e-4 - Use warmup to stabilize initial training - Monitor loss curves to adjust if needed - If loss explodes: reduce learning rate - If loss plateaus early: increase learning rate

Example:

```
training_args = Seq2SeqTrainingArguments(  
    learning_rate=5e-5,  
    lr_scheduler_type="linear",  
    warmup_steps=500,  
)
```

4. Weight Decay

Definition: L2 regularization that penalizes large weights to prevent overfitting.

Typical Range: 0.0 to 0.1

How It Works: Adds penalty term to loss function

$$\text{Loss_total} = \text{Loss_task} + \text{weight_decay} \times \sum(\text{weights}^2)$$

Guidelines: - **No weight decay (0.0):** No regularization - **Light (0.01):** Minimal regularization, common default - **Moderate (0.05):** Good for smaller datasets - **Heavy (0.1):** Strong regularization for overfitting prevention

Best Practices: - Start with 0.01 (common default) - Increase if overfitting occurs - Decrease if underfitting - Monitor validation metrics

```
weight_decay = 0.01 # L2 regularization strength
```

5. Gradient Clipping

Definition: Limits the maximum gradient value to prevent exploding gradients.

Parameter: `max_grad_norm`

Typical Value: 1.0

How It Works:

```
if gradient_norm > max_grad_norm:  
    gradient = gradient × (max_grad_norm / gradient_norm)
```

Purpose: - Prevents training instability - Stops gradient explosions - Particularly important for RNNs and long sequences

Best Practices: - Default value of 1.0 works well for most cases - Increase to 5.0 if you need more gradient freedom - Decrease to 0.5 for very stable training

```
max_grad_norm = 1.0 # Clip gradients above this norm
```

6. Dropout

Definition: Randomly drops (sets to zero) a percentage of neurons during training to prevent overfitting.

Typical Range: 0.0 to 0.3

Types: - **Attention Dropout:** Applied to attention weights - **Activation Dropout:** Applied to hidden states
- **Overall Dropout:** General dropout rate

Guidelines: - **No dropout (0.0):** No regularization - **Light (0.1):** Standard for well-sized datasets -
Moderate (0.2): Good for smaller datasets - **Heavy (0.3):** Aggressive overfitting prevention

Note: Dropout is only active during training, disabled during evaluation.

```
dropout = 0.1
attention_dropout = 0.1
```

Evaluation and Monitoring Parameters

7. Evaluation Strategy

Definition: How often to evaluate model on validation set.

Options:

```
evaluation_strategy = "steps"
eval_steps = 500 # Evaluate every 500 training steps

evaluation_strategy = "epoch"
```

Best Practices: - For small datasets: `evaluation_strategy="epoch"` - For large datasets: `evaluation_strategy="steps"` with `eval_steps=500-1000` - More frequent evaluation = better monitoring but slower training

8. Save Strategy

Definition: How often to save model checkpoints.

```
save_strategy = "steps" # or "epoch"
save_steps = 500 # Save every 500 steps
save_total_limit = 3 # Keep only best 3 checkpoints
```

Best Practices: - Match save strategy to evaluation strategy - Use `save_total_limit` to prevent disk space issues - Enable `load_best_model_at_end=True` for optimal final model

9. Logging

Definition: How often to log training metrics.

```
logging_steps = 100 # Log every 100 steps  
report_to = ["tensorboard"] # or "wandb", "none"
```

Advanced Parameters

10. Optimizer

Definition: Algorithm used to update model weights.

Common Options:

```
optim = "adamw_torch"  
  
optim = "adamw_8bit"  
  
optim = "adafactor"
```

Best Practice: Use AdamW for most cases

11. Mixed Precision Training

Definition: Uses lower precision (FP16/BF16) to speed up training and reduce memory.

```
fp16 = True # For older GPUs (Nvidia Volta, Turing)  
bf16 = True # For newer GPUs (Nvidia Ampere, Ada) - more stable
```

Benefits: - 2x faster training - 50% less memory usage - Minimal accuracy impact

12. Generation Parameters (for Seq2Seq)

For STT models during evaluation:

```
generation_max_length = 225  
  
generation_num_beams = 1 # Greedy decoding (fastest)  
generation_num_beams = 5 # Better quality, slower
```

Complete Example Configuration

```

from transformers import Seq2SeqTrainingArguments

training_args = Seq2SeqTrainingArguments(
    # Output
    output_dir="../whisper-finetuned",

    # Training duration
    num_train_epochs=10,

    # Batch sizes
    per_device_train_batch_size=8,
    per_device_eval_batch_size=16,
    gradient_accumulation_steps=4, # Effective batch size = 32

    # Learning rate
    learning_rate=5e-5,
    lr_scheduler_type="linear",
    warmup_steps=500,

    # Regularization
    weight_decay=0.01,
    max_grad_norm=1.0,

    # Evaluation
    evaluation_strategy="steps",
    eval_steps=500,

    # Saving
    save_strategy="steps",
    save_steps=500,
    save_total_limit=3,
    load_best_model_at_end=True,

    # Logging
    logging_steps=100,
    report_to=["tensorboard"],

    # Performance
    fp16=True, # or bf16=True for newer GPUs

    # Generation (for evaluation)
    predict_with_generate=True,
    generation_max_length=225,
    generation_num_beams=1,

    # Optimization
    optim="adamw_torch",

    # Misc
    push_to_hub=False,

```

```
    metric_for_best_model="wer", # Word Error Rate  
    greater_is_better=False, # Lower WER is better  
)
```

Parameter Tuning Guidelines

Starting Point (Conservative)

```
num_train_epochs=5  
per_device_train_batch_size=8  
learning_rate=1e-5  
warmup_steps=500  
weight_decay=0.01
```

For Small Datasets (< 20 hours)

```
num_train_epochs=3-5  
per_device_train_batch_size=4-8  
learning_rate=1e-5  
weight_decay=0.05 # Higher regularization  
dropout=0.2
```

For Large Datasets (> 100 hours)

```
num_train_epochs=10-20  
per_device_train_batch_size=16-32  
learning_rate=5e-5  
weight_decay=0.01  
warmup_steps=1000
```

If Overfitting

```
num_train_epochs -= 2

weight_decay += 0.02
dropout += 0.1

early_stopping_patience=3
```

If Underfitting

```
num_train_epochs += 5

learning_rate *= 2

weight_decay /= 2
```

Monitoring Guidelines

Track these metrics during training:

1. **Training Loss:** Should steadily decrease
2. **Validation Loss:** Should decrease and track training loss
3. **WER (Word Error Rate):** Should steadily decrease
4. **Learning Rate:** Check schedule is working as expected
5. **Gradient Norm:** Should be stable, not exploding

Red Flags: - Validation loss increases while training loss decreases → Overfitting - Both losses plateau early → Underfitting or learning rate too low - Loss becomes NaN → Gradient explosion (reduce LR or clip gradients) - No improvement after several epochs → Hyperparameter adjustment needed

Summary Table

Parameter	Typical Range	Purpose	Adjustment Strategy
Epochs	3-20	Training duration	Monitor validation loss
Batch Size	4-32	Memory/speed trade-off	Maximize within GPU limits
Learning Rate	1e-5 to 1e-4	Update speed	Reduce if unstable
Weight Decay	0.0-0.1	Regularization	Increase if overfitting
Warmup Steps	500-2000	Training stability	5-10% of total steps
Gradient Clipping	1.0	Prevent explosions	Keep at 1.0 usually
Dropout	0.0-0.3	Regularization	Increase if overfitting

Conclusion

Successful fine-tuning requires careful balancing of these parameters. Start with conservative defaults, monitor validation metrics closely, and adjust based on training behavior. Remember that every dataset is different, so experimentation and iteration are key to achieving optimal results.

Part V: Inference & Deployment

Running and deploying ASR models

30 Second Chunking Whisper Streaming

Question

Why is there a 30-second limit in Whisper transcription (especially on mobile)? Does it make a meaningful difference whether you use “voice typing” (real-time chunking) versus “async transcription” (recording then transcribing in one go)? Are the processing architectures different, and do you get different results?

Answer

Excellent question that touches on a fundamental Whisper architecture constraint and how it manifests in practical applications. Let’s unpack this.

Why the 30-Second Limit Exists

Whisper’s Architecture Constraint

Whisper has a hard 30-second input window by design, set during training:

Technical Details:

```
Audio Input → Mel-spectrogram (80 bins)  
Time dimension: Fixed at 3000 frames (30 seconds at 100 fps)  
Positional encodings: Fixed to 1500 positions (encoder + decoder)
```

Whisper cannot process audio >30 seconds without chunking.

Why 30 Seconds?

1. **Training Data:** Most training samples were subtitle segments (<30s)
2. **Memory:** Transformer attention is $O(n^2)$ —longer sequences = exponential memory
3. **Positional Encodings:** Fixed-length learned positional embeddings (not relative)
4. **Balance:** 30s is long enough for context, short enough for efficiency

This is NOT a mobile limitation—it's a fundamental model constraint.

Desktop Whisper has the same 30-second limit. Mobile inherits it because they're using the same model architecture.

How Applications Handle >30-Second Audio

When you transcribe longer audio, there are two common approaches:

Approach 1: Sequential Chunking (What You're Experiencing)

How It Works:

```
Audio (5 minutes) → Split into 30s chunks → Process chunk 1 → chunk 2 → ... →  
chunk 10
```

Implementation (Typical Mobile App):

```

def transcribe_long_audio(audio_file):
    chunks = split_audio_30s(audio_file)
    transcriptions = []

    for chunk in chunks:
        result = whisper.transcribe(chunk) # Each takes 2-5 seconds
        transcriptions.append(result)

    return " ".join(transcriptions)

```

What You're Noticing: - Processing happens **sequentially** (one chunk at a time) - There's a delay/stutter at 30s boundaries - Each chunk is independent (no context from previous chunks)

Problems: 1. **Boundary Issues:** Words/sentences split at 30s mark → transcription errors 2. **Sequential Latency:** Each chunk takes 2-5s → 5min audio = 10 chunks × 3s = 30s processing 3. **Context Loss:** Chunk 2 doesn't know what was said in chunk 1

Approach 2: Overlapping Chunking (Better, But Rarer)

How It Works:

```

Chunk 1: [0-30s]
Chunk 2: [25-55s] ← 5-second overlap
Chunk 3: [50-80s] ← 5-second overlap
...

```

Benefits: - Overlap ensures words at boundaries are fully captured - Can merge overlapping transcriptions intelligently - Reduces boundary errors

Drawbacks: - More chunks to process (slightly slower) - Need smarter merging logic

Few mobile apps implement this (more complex code).

Voice Typing vs. Async Transcription: Key Differences

Voice Typing (Real-Time / Streaming)

How It Works:

You speak → App captures 30s → Processes → Displays text → Captures next 30s
→ ...

Implementation Details: - **Live audio buffer:** Continuously recording - **Trigger at 30s:** When buffer fills, send to Whisper - **Display immediately:** Show text as it's transcribed - **Next chunk:** Start new buffer while displaying previous result

User Experience: - Text appears in ~30-second bursts - Noticeable pauses at 30s boundaries (processing delay) - Can't go back and correct later chunks based on earlier context

Pros: - Immediate feedback (see text as you speak) - Good for short dictation (emails, messages)

Cons: - Stuttering at boundaries - Higher cognitive load (watching text appear) - Boundary errors more noticeable (mid-sentence splits)

Async Transcription (Record Then Transcribe)

How It Works:

You speak (5 min) → Record entire audio → Send for transcription → Process all chunks → Return full text

Implementation Details: - **Record full audio:** Capture entire note/recording - **Save as single file:** WAV, MP3, etc. - **Chunk at processing time:** Split into 30s segments when transcribing - **Process in batch:** Can use parallel processing (if hardware supports)

User Experience: - No live feedback while speaking - Processing happens all at once after recording - Get complete transcription result

Pros: - Better for long-form (lectures, meetings, notes) - Can optimize chunking (overlapping, silence detection) - Parallel processing possible (faster on multi-core) - Can add post-processing (punctuation, paragraphs)

Cons: - No live feedback (don't know if it's working) - All-or-nothing (if it fails, lose everything)

Does It Make a Meaningful Difference?

Short Answer: Yes, but nuanced.

Aspect	Voice Typing	Async Transcription
Accuracy	Same (model is identical)	Same (model is identical)
Boundary Errors	More noticeable	Can be reduced with overlap
Processing Speed	Perceived slower (sequential + waiting)	Can be faster (batch + parallel)
User Experience	Choppy, stuttering	Smooth, all-at-once
Best For	Short dictation (<2 min)	Long notes (>2 min)

Accuracy: Mostly the Same

If both approaches use **sequential chunking without overlap**, accuracy will be identical: - Same model - Same chunks - Same transcription per chunk

However, async transcription CAN be more accurate if: 1. **Overlapping chunks**: Reduces boundary errors 2. **Smart segmentation**: Chunks split at pauses, not arbitrary 30s 3. **Post-processing**: Can apply punctuation/paragraph models on full text

Performance: Async Can Be Faster

Voice Typing (Serial Processing):

```
Speak 30s → Wait 3s (processing) → Speak 30s → Wait 3s → ...  
Total time: 5 min speaking + 30s processing = 5:30 total
```

Async (Batch Processing):

```
Speak 5 min → Process all 10 chunks in parallel (if multi-core) → 3-5s total  
Total time: 5 min speaking + 5s processing = 5:05 total
```

But your phone (OnePlus Nord 3) likely does NOT parallelize (APU may not support it, or app doesn't implement it), so async is processed sequentially anyway:

Speak 5 min → Process chunks 1-10 sequentially → 30s processing
Total time: 5 min speaking + 30s processing = 5:30 total

So performance is similar for your hardware unless the app is highly optimized.

Boundary Handling: Async Can Be Better

Voice Typing:

[Chunk 1]: "...and then we decided to go to the st-"
[Chunk 2]: "ore to buy some groceries"

Result: "st ore" (word split, likely transcription error)

Async with Overlapping:

[Chunk 1]: "...and then we decided to go to the st-"
[Overlap]: "to the store to buy" ← captures full word
[Chunk 2]: "ore to buy some groceries"

Merge: "...and then we decided to go to the store to buy some groceries"

Result: Correct transcription

Most mobile apps don't do overlapping, so this advantage is theoretical unless you use a sophisticated app.

Practical Implications for Your Use Case

Your Observation: “Choppy Process” Around 30s Mark

What's Happening: 1. At ~29 seconds: App prepares to send chunk to Whisper 2. At 30 seconds: Processing starts (2-5 second delay) 3. During processing: Either - Audio recording pauses (you can't speak) → **very choppy** - Audio recording continues but processing blocks UI → **laggy**

This is a real-time processing bottleneck, not inherent to Whisper.

Solution: - **Better apps:** Buffer next chunk while processing previous (seamless) - **Async transcription:** Avoid this issue entirely (no live processing)

Which Approach Should You Use?

For Note-Taking (Your Primary Use Case):

Recommendation: Async Transcription

Why: 1. **Better accuracy:** Can use overlapping chunks 2. **No interruptions:** Record full thought without pauses 3. **Post-processing:** Can apply punctuation/paragraph tools after 4. **Less frustrating:** No choppy 30s boundaries

Implementation: - Use a voice recorder app (record full note) - Transcribe afterward using: - Desktop (Faster-Whisper with overlapping) - Mobile app that supports async (SpeechNote, others)

For Short Dictation (Messages, Emails):

Voice typing is fine (<2 minutes, a few chunks).

Best of Both Worlds:

Use a hybrid approach: 1. **Short inputs (<1 min):** Voice typing for immediacy 2. **Long inputs (>2 min):** Async transcription for quality

Optimizing Async Transcription on Your Setup

On Desktop (AMD 7700 XT):

Use **Faster-Whisper with overlapping**:

```

from faster_whisper import WhisperModel

model = WhisperModel("medium", device="cuda", compute_type="float16")

segments, info = model.transcribe(
    "long_note.wav",
    vad_filter=True, # Voice Activity Detection (skip silence)
    vad_parameters=dict(
        min_silence_duration_ms=500, # Chunk at pauses
    )
)

full_transcription = " ".join([seg.text for seg in segments])

```

Benefits: - VAD (Voice Activity Detection) chunks at natural pauses (not arbitrary 30s) - Faster processing (CTranslate2 engine) - Better boundary handling

On Phone (OnePlus Nord 3):

Option 1: Record + Upload to Desktop

Record on phone → Transfer to desktop → Transcribe with Faster-Whisper

Best accuracy, but requires transfer step.

Option 2: Use App with Smart Chunking Look for Android apps that support:
- Overlapping chunks
- VAD-based segmentation
- Post-processing

Candidates: - **SpeechNote** (Linux, but check Android version features) - **Whisper.cpp-based apps** (some support smart chunking) - **Transcription tools with VAD**

The Underlying Question: Can We Remove the 30s Limit?

Short answer: Not with current Whisper architecture.

Future Models: - **Relative positional encodings:** Could support arbitrary length - **Sliding window transformers:** Process long audio in overlapping windows - **Chunking-aware training:** Train models specifically to handle chunks better

Current Research: - **Whisper-Longformer:** Experimental variants with longer context - **Streaming Whisper:** Optimized for real-time with better boundary handling

But for now, 30-second chunking is unavoidable with Whisper.

Summary

Question	Answer
Why 30s limit?	Whisper's architecture (fixed positional encodings, memory constraints)
Voice typing vs. async: different architectures?	No—both use same chunking, but async can optimize better
Meaningfully different results?	Accuracy: same. UX: async is better for long-form
Recommend for note-taking?	Async transcription with overlapping/VAD

The “choppy” experience you’re noticing is a **real-time processing UX issue**, not fundamental to Whisper. Async transcription (record → transcribe) avoids this and allows for better optimization (overlapping chunks, VAD, post-processing).

For your use case (note-taking, longer recordings), async transcription is superior.

Note: This explanation was generated by Claude Code (claude-sonnet-4-5) for Daniel Rosehill’s STT Fine-Tuning Notebook. Whisper’s 30-second limit is architectural and unlikely to change in current versions. For production note-taking workflows, consider using Faster-Whisper on desktop with VAD-based chunking for best results, or mobile apps that implement intelligent segmentation. Always test both approaches with your specific audio to verify practical differences.

Deployment Options For Custom Asr

Question Summary

Daniel is exploring deployment options for fine-tuned or custom ASR models, particularly for individual/solo users. He's found Replicate for serverless but is concerned about costs for 24/7 operation. He wants to understand the full spectrum of deployment options and cost implications for both serverless and always-on (local or cloud) deployments.

Answer

You're right that this is somewhat niche territory for individual users, but it's increasingly relevant as more people fine-tune their own ASR models. Let me break down the deployment landscape comprehensively.

Serverless Inference Options

1. Replicate - What you found: Yes, Replicate is the most prominent serverless option - **Pricing:** Pay-per-second of inference time - Typically \$0.0005-0.0025 per second depending on hardware (CPU vs GPU) - For Whisper-sized models on GPU: ~\$0.001/second - **Cost Example:** - 1 hour of audio processing ≈ 6 minutes inference time (10x realtime) - Cost: ~\$0.36 per hour of audio transcribed - For intermittent use (say, 5 hours of audio/month): ~\$1.80/month - **Pros:** Zero setup, scales automatically, no idle costs - **Cons:** Cold start latency (2-15 seconds), per-request costs add up quickly for heavy use

2. Hugging Face Inference Endpoints - Overview: Serverless inference for models hosted on HuggingFace - **Pricing Tiers:** - Free tier: Limited requests, public models only - Paid: \$0.06/hour (CPU) to \$1.50/hour (GPU) when running - Auto-scales to zero when idle (no requests for 15 minutes) - **Cost Example:** - If processing requests sporadically (active 2 hours/day): ~\$90/month for GPU instance - Better than 24/7 (\$1,080/month) but still pricey for continuous use - **Pros:** Good HuggingFace integration, custom model support - **Cons:** Not truly serverless (charges per hour active, not per request)

3. Modal - Overview: Python-native serverless compute platform - **Pricing:** Pay per GPU-second - A10G GPU: ~\$0.0010/second - T4 GPU: ~\$0.0005/second - **Cost Example:** - Processing 10 hours of audio/day (realtime inference): ~\$36/month on T4 - **Pros:** Excellent developer experience, true pay-per-use, fast cold starts - **Cons:** Requires some Python infrastructure code setup

4. Banana.dev (now Tonic.ai) - Overview: Serverless GPU inference platform - **Pricing:** Similar to Replicate (~\$0.0008/second for GPU) - **Status:** Rebranded/transitioning, may be less stable option currently - **Pros:** Previously popular for ASR deployments - **Cons:** Platform uncertainty after rebrand

5. Baseten - Overview: ML inference platform with serverless and dedicated options - **Pricing:** Custom pricing, typically \$0.0005-0.0015/second - **Pros:** Good performance, handles custom models well - **Cons:** Less transparent pricing, requires contact for details

6. AWS Lambda + GPU (Emerging) - Overview: AWS is rolling out Lambda support for GPUs - **Status:** Limited availability, not yet widely practical for ASR - **Future Potential:** Could become very cost-effective for sporadic use

24/7 Self-Hosted Options

If you want always-available inference (locally or cloud), here are the realistic options:

Local Deployment (Home Server)

Option A: Dedicated Machine - Hardware Requirements for Whisper: - CPU-only: Modern 8-core CPU (i7/Ryzen 7), 16GB RAM - GPU: RTX 3060 (12GB VRAM) or better for comfortable performance - Storage: 50-100GB SSD for models and OS

- **Costs:**

- **Initial:** \$800-1,500 for dedicated machine (or use existing hardware)

- **Electricity:**

- Idle GPU server: $\sim 100-150\text{W} = \sim \$10-15/\text{month}$ (at \$0.12/kWh)
 - Under load: $\sim 250\text{W} = \sim \$25/\text{month}$
 - Annual: $\sim \$120-300/\text{year}$ in electricity

- **Networking:**

- Port forwarding: Free (security risk - need VPN)
 - Cloudflare Tunnel: Free (recommended, secure)
 - Tailscale/ZeroTier: Free for personal use (private network)

Option B: Your Existing Hardware - You have AMD RX 7700 XT with ROCm - excellent for ASR! - **Costs:** - Electricity only (~\$10-20/month if running 24/7) - Wear and tear on GPU (negligible for inference) - **Pros:** No additional hardware cost, full control - **Cons:** Home network dependency, potential security exposure

Recommended Setup for Local 24/7:

Cloud VPS Deployment

Option 1: CPU-Only VPS (Budget) - Providers: Hetzner, OVH, DigitalOcean, Linode - **Recommended Specs:** 8-core CPU, 16GB RAM - **Costs:** - Hetzner CCX33: €32.69/month (~\$35/month) - 8 vCores, 32GB RAM - DigitalOcean: \$48/month - 8 vCPU, 16GB RAM - **Performance:** - Realtime or slightly faster for Whisper-large - Acceptable for most use cases - **Pros:** Predictable costs, reliable, no home network dependency - **Cons:** Slower than GPU inference

Option 2: GPU Cloud Instances - RunPod: - RTX A4000 (16GB): ~\$0.34/hour = ~\$245/month for 24/7 - RTX 4090 (24GB): ~\$0.69/hour = ~\$497/month for 24/7 - **Vast.ai:** - RTX 3060 (12GB): ~\$0.15/hour = ~\$108/month for 24/7 - Highly variable pricing (spot market) - **Lambda Labs:** - A10 GPU: \$0.60/hour = ~\$432/month - **Google Cloud / AWS / Azure:** - Much more expensive (~\$0.70-2.00/hour for GPU instances) - GCP T4: ~\$0.35/hour = ~\$252/month

Option 3: Hybrid Approach (Spot Instances) - Vast.ai Spot Instances: - Bid on idle GPU capacity - Can get RTX 3080 for ~\$0.10/hour = ~\$72/month - Risk: Instance can be reclaimed (need auto-restart logic) - **AWS Spot / GCP Preemptible:** - 60-80% cheaper than on-demand - Requires interruption handling

Cost Comparison Summary

Deployment Option	Setup Cost	Monthly Cost (Light Use)	Monthly Cost (Heavy/24-7)
Replicate	\$0	\$5-20	\$300-1,000+
Modal	\$0	\$10-50	\$200-500
HF Inference Endpoints	\$0	\$30-100	\$1,080 (GPU always-on)
Local (Existing HW)	\$0	\$10-20	\$15-30
Local (New Server)	\$800-1,500	\$10-20	\$15-30
CPU VPS (Hetzner)	\$0	\$35	\$35
GPU Cloud (Vast.ai)	\$0	\$108+	\$108-500
GPU Cloud (RunPod)	\$0	\$245+	\$245-500

Recommendations Based on Use Cases

Scenario 1: Occasional Personal Use (< 10 hours audio/month) - **Best Option:** Replicate or Modal - **Reasoning:** Zero setup, only pay for what you use - **Cost:** \$5-20/month

Scenario 2: Regular Personal Use (Daily, ~2-4 hours audio/day) - **Best Option:** Local deployment on your existing hardware - **Reasoning:** Electricity costs less than serverless, full control - **Cost:** ~\$15-25/month (electricity only) - **Setup:** Docker + Faster-Whisper + Cloudflare Tunnel

Scenario 3: Service/App Development (Public API) - **Best Option:** CPU VPS (Hetzner) with queue system - **Reasoning:** Predictable costs, good performance, professional reliability - **Cost:** ~\$35-50/month - **Alternative:** Modal for burst capacity + CPU VPS for base load

Scenario 4: High-Volume Production (100+ hours audio/day) - **Best Option:** Dedicated GPU cloud (RunPod/Vast.ai) or multiple CPU VPS - **Reasoning:** Cost-effective at scale - **Cost:** \$250-500/month

Your Specific Situation (Solo User, Custom Model)

Given your setup (AMD GPU with ROCm), here's what I'd recommend:

Option A: Local 24/7 (Recommended)

- Zero additional hardware cost (**you have RX 7700 XT**)
 - Whisper runs well on ROCm (**HSA_OVERRIDE_GFX_VERSION=11.0.1**)
 - Can expose via Cloudflare Tunnel (**free**, secure, no port forwarding)
 - Total cost: **~\$15-20/month in electricity**
1. Docker container with whisper.cpp or faster-whisper
 2. FastAPI wrapper for REST API
 3. Cloudflare Tunnel for secure external access
 4. Optional: Nginx reverse proxy for API management

Option B: Hybrid (Local + Serverless Fallback)

Option C: CPU VPS (If You Don't Want Local Running 24/7)

Practical Cost Calculation Examples

Scenario: Processing 5 hours of audio per day

Option	Daily Cost	Monthly Cost	Notes
Replicate (10x RT)	\$1.80	\$54	Quick bursts
Modal (realtime)	\$1.20	\$36	Python-friendly
Local (Your GPU)	\$0.50	\$15	Electricity only
Hetzner CPU VPS	\$1.10	\$33	Always available
Vast.ai GPU (spot)	\$2.40	\$72	Fast processing

Verdict for Solo User: Local deployment on your existing hardware is by far the most cost-effective for 24/7 availability.

Exposure/Security Considerations

If running locally and exposing to internet:

1. **Never expose ports directly** - major security risk

2. **Use Cloudflare Tunnel** (recommended):

```
# Free, secure, no port forwarding needed
cloudflared tunnel create my-asr
# Creates encrypted tunnel from your server to Cloudflare edge
```

3. **Alternative: Tailscale** - Private mesh network (free for personal use)

4. **API Authentication:** Always implement API keys/tokens

5. **Rate Limiting:** Prevent abuse with request limits

6. **HTTPS Only:** Cloudflare provides this automatically

Advanced Options for Solo Users

Option: Fly.io - Deploy containers globally - Pay per request (scales to zero) - ~\$0.0008/sec GPU or \$0.00025/sec CPU - Good middle ground between VPS and serverless

Option: Railway.app - \$5/month base + usage - Good for hobby projects - No GPU support (CPU only)

Option: Self-hosted on Oracle Cloud Free Tier - 4 ARM cores, 24GB RAM - completely free forever - Can run CPU inference - Performance: Slower than x86, but usable for Whisper-base/small - Great for experimentation

Final Recommendation for You

Based on your setup and likely use pattern:

1. **Start with local deployment** on your RX 7700 XT
 - Use Docker + faster-whisper with ROCm
 - Expose via Cloudflare Tunnel
 - Cost: ~\$15-20/month electricity
 - Benefit: Full control, lowest cost, instant inference
2. **Add Modal as backup** for when local is down
 - Minimal cost if rarely used
 - Python-friendly deployment
 - Automatic fallback logic in your client
3. **If you outgrow local:** Migrate to Hetzner CPU VPS
 - Still cheaper than GPU cloud options
 - Professional reliability
 - ~\$35/month predictable cost

Code Example: Local Deployment with Cloudflare Tunnel

```
wget -q https://github.com/cloudflare/cloudflared/releases/latest/download/
      cloudflared-linux-amd64.deb
sudo dpkg -i cloudflared-linux-amd64.deb

cloudflared tunnel login

cloudflared tunnel create my-asr-api

cat > ~/.cloudflared/config.yml << EOF
tunnel: my-asr-api
credentials-file: /home/daniel/.cloudflared/<tunnel-id>.json

ingress:
- hostname: asr.yourdomain.com
  service: http://localhost:8000
- service: http_status:404
EOF

sudo cloudflared service install
sudo systemctl start cloudflared
```

Conclusion

For a solo user with a custom ASR model:

- **Serverless options exist beyond Replicate** (Modal, HF Inference Endpoints, Baseten, Fly.io)
- **Local 24/7 deployment is surprisingly affordable** (~\$15-30/month) using existing hardware
- **Cloud VPS CPU instances** are the sweet spot for “always-on” without managing local hardware (~\$35/month)
- **GPU cloud is expensive for 24/7** (\$250-500/month) but reasonable for burst usage
- **Your specific situation:** Local deployment on RX 7700 XT + Cloudflare Tunnel is the optimal approach

The niche you’re in (solo user with custom ASR) actually has more options than you might think, and costs can be quite reasonable with the right architecture!

Note: This response was generated by Claude Code as part of Daniel’s STT Fine-Tuning Notebook project. Pricing information is approximate and based on 2025 rates. Always verify current pricing with service providers.

Live Vs Batch Transcription

Overview

ASR systems typically operate in two distinct modes:

1. **Live/streaming transcription:** Real-time transcription as you speak, with text appearing incrementally
2. **Batch/file transcription:** Upload a complete audio file and receive the full transcription after processing

While these often use the same underlying model (e.g., Whisper), there are significant architectural and accuracy differences between these approaches.

Architectural Differences

Live/Streaming Transcription

How it works:

1. **Audio buffering:** Audio is captured in small chunks (typically 0.5-3 seconds)
2. **Continuous processing:** Each chunk is processed as it arrives, with minimal delay
3. **Context windowing:** The model maintains a sliding context window, using previous chunks to inform current transcription
4. **Incremental output:** Text appears progressively as each chunk is transcribed
5. **Voice Activity Detection (VAD):** System detects when you're speaking vs silent to determine chunk boundaries

Technical implementation:

```
Audio stream → VAD → Chunking (0.5-3s) → Buffering → Model inference → Text output  
↓  
Context window (previous 30s typically maintained)
```

Constraints:

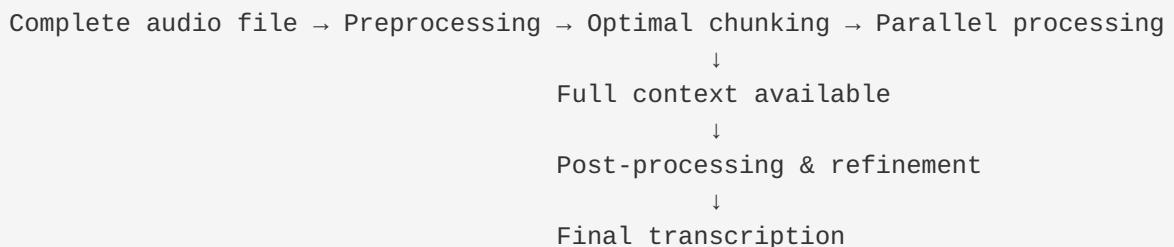
- **Fixed latency requirements:** Must process within ~100-500ms to feel “real-time”
- **Limited context:** Can only look back at recent audio (typically 30 seconds maximum)
- **No future context:** Cannot see what comes next in the sentence
- **Chunk boundary issues:** Must make decisions about incomplete phrases
- **Computational pressure:** Must process continuously without falling behind

Batch/File Transcription

How it works:

1. **Complete file upload:** Entire audio file is available before processing begins
2. **Preprocessing:** Can apply audio normalization, noise reduction, and enhancement to the entire file
3. **Optimal segmentation:** Can analyze the entire audio to find natural boundaries (pauses, speaker changes)
4. **Full context processing:** Model can use surrounding context from the entire recording
5. **Multi-pass processing:** Can make multiple passes over ambiguous sections
6. **Post-processing:** Can apply additional cleanup, punctuation restoration, and confidence-based corrections

Technical implementation:



Advantages:

- **No latency constraints:** Can take as long as needed for optimal results
- **Full bidirectional context:** Can look both backward and forward
- **Better segmentation:** Can find optimal chunk boundaries after analyzing the whole file
- **Multiple passes:** Can revisit uncertain sections with more context
- **Better preprocessing:** Can apply sophisticated audio enhancement knowing the full characteristics

Why Batch Transcription Often Performs Better

The perception that batch transcription is more accurate is **not imagination**—there are real technical reasons:

1. Bidirectional Context

- **Live:** Can only look backward (previous 30 seconds typically)
- **Batch:** Can look both backward AND forward
- **Impact:** Understanding upcoming context helps disambiguate current words (e.g., knowing someone will say “bank account” vs “river bank”)

2. Optimal Chunk Boundaries

- **Live:** Must chunk based on real-time constraints, sometimes cutting mid-phrase
- **Batch:** Can analyze entire audio to find natural pauses and segment at optimal points
- **Impact:** Models perform better when chunks align with linguistic boundaries (sentence/phrase endings)

3. Audio Preprocessing

- **Live:** Limited preprocessing (simple noise gating, maybe basic noise reduction)
- **Batch:** Can analyze full audio characteristics and apply:
 - Sophisticated noise profiling and removal
 - Dynamic range compression optimized for the specific recording
 - Spectral enhancement tuned to the speaker’s voice characteristics
- **Impact:** Cleaner audio input = better transcription accuracy

4. No Pressure for Real-Time Performance

- **Live:** Must use faster, sometimes less accurate inference settings
- **Batch:** Can use slower, more accurate inference parameters:
 - Higher beam search width
 - More sophisticated language model scoring
 - Temperature sampling for better alternatives
- **Impact:** 5-15% accuracy improvement possible with more computational resources

5. Error Correction Opportunities

- **Live:** Text is output immediately, limited ability to revise
- **Batch:** Can apply post-processing:
 - Confidence-based revision
 - Language model rescoring
 - Consistency checking across the full transcript
- **Impact:** Can catch and correct errors that seem wrong in broader context

6. Speaker Adaptation

- **Live:** Limited adaptation in first 30-60 seconds
- **Batch:** Can analyze the entire recording first to:
 - Identify speaker characteristics
 - Build speaker-specific acoustic model adjustments
 - Learn vocabulary and speaking patterns used throughout
- **Impact:** Better performance on uncommon pronunciations and speaking styles

API Architecture Differences

Most ASR service providers (OpenAI, AssemblyAI, Deepgram, etc.) use **different endpoints** for live vs batch:

Streaming Endpoints

- Use WebSocket connections for bidirectional communication
- Implement different inference optimizations (speed over accuracy)
- May use lighter model variants
- Limited preprocessing capabilities
- Stateful connections with context management

Batch Endpoints

- Use standard HTTP POST with file upload
- Implement full inference optimizations (accuracy over speed)

- May use larger/better model variants
- Full preprocessing pipeline
- Stateless processing with full context available

The 15-Minute Recording Scenario

Let's compare your two approaches for a 15-minute recording:

Approach 1: Live transcription with 30-second chunks

What happens: - Audio captured in ~30 half-second chunks - Each chunk processed with context from previous ~30 seconds - Model makes ~30 independent inference decisions - Text appears progressively - Total processing: 15 minutes of real-time processing

Accuracy factors: - ✗ Forward context not available - ✗ Chunk boundaries not optimized - ✗ Limited preprocessing - ✗ Fast inference parameters required - ✗ No multi-pass opportunities

Approach 2: Record in Audacity → upload MP3 → transcribe

What happens: - Complete 15-minute audio file available - System analyzes full audio for characteristics - Optimal chunk boundaries identified (perhaps 60-90 chunks at natural pauses) - Each chunk processed with full recording context - Post-processing applied to final transcript - Total processing: 1-3 minutes

Accuracy factors: - ✓ Full bidirectional context - ✓ Optimized chunk boundaries - ✓ Full preprocessing applied - ✓ Optimal inference parameters - ✓ Post-processing applied

Expected accuracy difference: 5-20% word error rate improvement, depending on audio quality and content complexity

When Live Transcription Makes Sense

Despite the accuracy tradeoffs, live transcription is valuable for:

1. **Interactive applications:** Dictation, voice commands, live captions
2. **Immediate feedback needs:** Making corrections while recording
3. **Long recordings:** Don't want to wait 2 hours for a 2-hour meeting
4. **Memory constraints:** Can't store entire large audio file

5. **Privacy concerns:** Don't want to upload complete files

Recommendations for Best Results

For Live Transcription:

1. **Use models optimized for streaming:** Some Whisper variants are specifically tuned for streaming
2. **Ensure good audio quality:** Use quality microphone, quiet environment
3. **Speak clearly with pauses:** Help the VAD and chunking
4. **Use longer context windows:** If supported (e.g., 45-60 seconds vs 30)
5. **Consider hybrid approaches:** Live transcription with post-recording refinement pass

For Batch Transcription:

1. **Use highest quality audio:** Record at 16kHz+ sample rate, minimal compression
2. **Include silence at start/end:** Helps with processing boundary issues
3. **Use lossless formats when possible:** WAV/FLAC better than MP3
4. **Segment very long files:** Break multi-hour recordings into 30-60 minute segments
5. **Use provider's best quality tier:** Most services offer "fast" vs "accurate" tiers

Technical Deep Dive: Chunking in Live Transcription

Under the hood during live transcription:

```

audio_buffer = []
context_window = []

while recording:
    # Capture audio chunk (e.g., 30ms)
    chunk = capture_audio(30ms)
    audio_buffer.append(chunk)

    # When buffer reaches processing size (e.g., 1 second)
    if len(audio_buffer) >= processing_size:
        # Combine with context window
        input_audio = context_window + audio_buffer

        # Run inference
        transcription = model.transcribe(input_audio)

        # Output new text
        output(transcription.new_text)

        # Update context window (sliding window)
        context_window = audio_buffer[-context_size:]

    # Clear buffer
    audio_buffer = []

```

Key points:

- The model isn't truly processing "live"—it's processing discrete chunks rapidly
- Context window maintains recent audio for better accuracy
- Each inference sees only current chunk + recent context
- Decisions are made incrementally and can't easily be revised

Conclusion

Yes, batch transcription generally provides better accuracy than live transcription due to:

- Full bidirectional context
- Optimal preprocessing and segmentation
- Ability to use more sophisticated inference parameters
- Post-processing opportunities
- No real-time latency constraints

For your 15-minute recording scenario, recording in Audacity and uploading will almost certainly produce more accurate results than live transcription, typically with 5-20% better word error rates, especially for:

- Technical terminology
- Proper nouns
- Ambiguous words that need sentence context
- Challenging audio conditions

The tradeoff is waiting for processing rather than getting immediate feedback, but if accuracy is the priority, batch processing is the better choice.

Note: This document was generated by Claude Code, an AI assistant. Please validate technical details and test recommendations in your specific environment before implementing.

Second Gpu For Stt Workloads

Question Summary

Daniel is considering hardware upgrade strategies for his desktop setup. He currently uses an AMD GPU (chosen before getting into AI/STT work) and primarily uses speech-to-text throughout his workday. He's evaluating two options: 1. Replace the AMD GPU with an NVIDIA GPU 2. Keep the AMD GPU for display output and add a second GPU dedicated to STT processing

The question extends to whether a GPU, NPU, or TPU would be optimal for STT-only workloads, given that local LLM usage is minimal.

Recommendation: Strategic Hardware Considerations

Primary Recommendation: Single NVIDIA GPU (Next Build)

For your next build or major upgrade, I'd recommend **replacing your AMD GPU with a single NVIDIA GPU** rather than adding a second GPU. Here's why:

Advantages: - **Simplified system management:** One GPU means less power consumption, heat, driver complexity, and PCIe lane allocation issues - **CUDA ecosystem dominance:** The vast majority of AI/ML tools (including STT) have first-class CUDA support with better optimization - **Flexibility:** A single NVIDIA GPU can handle both display and AI workloads efficiently - **Better per-dollar performance:** You get more AI performance for your money with a single higher-tier NVIDIA card than splitting budget across two GPUs - **Lower power draw:** Modern NVIDIA GPUs (especially 4000 series) are remarkably power-efficient for AI workloads

Recommended GPU Tiers for STT + Light LLM:

1. **Budget Option (~\$500-600):** NVIDIA RTX 4060 Ti 16GB
 - 16GB VRAM is crucial for larger Whisper models and future-proofing
 - Excellent for STT inference (Whisper large-v3 runs smoothly)
 - Can handle local LLMs up to 13B parameters reasonably well
 - Low power consumption (~160W TDP)
2. **Mid-Range Sweet Spot (~\$800-1000):** NVIDIA RTX 4070 Ti / 4070 Ti Super
 - 12GB VRAM (4070 Ti) or 16GB VRAM (4070 Ti Super)
 - Significantly faster inference for Whisper
 - Better headroom for local LLM experimentation
 - Still reasonable power draw (~285W TDP)
3. **High-End Option (~\$1200-1500):** NVIDIA RTX 4080 / 4080 Super
 - 16GB VRAM
 - Overkill for STT alone, but excellent for any AI workload you might explore
 - Near-workstation performance for AI tasks

Why Not a Second GPU?

Technical Drawbacks: - **PCIe lane limitations:** Most consumer motherboards don't have enough PCIe lanes to run two GPUs at full bandwidth, meaning you'd likely run both at x8 instead of x16 - **Power supply requirements:** You'd need a significantly larger PSU (likely 1000W+) - **Heat and cooling:** Two GPUs generate substantial heat; your case might not have adequate cooling - **Driver complexity:** Running AMD for display + NVIDIA for compute adds driver management overhead - **ROCm limitations:** Your current AMD GPU already struggles with ROCm support for AI (as you've likely experienced), so keeping it doesn't provide much benefit

Cost Consideration: A mid-range NVIDIA GPU (~\$800) would likely provide better AI performance than your current AMD GPU + a budget NVIDIA card costing the same total amount.

GPU vs NPU vs TPU for STT

GPU (Recommended for STT): - ✅ Best option for STT workloads - ✅ Whisper and similar models are heavily optimized for GPU - ✅ Flexibility for other AI tasks (image generation, LLMs) - ✅ Mature software ecosystem (PyTorch, ONNX, faster-whisper, CTranslate2)

NPU (Neural Processing Unit): - ❌ Not recommended for desktop STT - NPUs are designed for low-power inference on mobile/edge devices - Poor software support for Whisper models on NPUs - Would require significant model conversion/quantization work - Performance would likely be worse than GPU for your use case - Examples: Intel's AI Boost, Qualcomm's Hexagon NPU (laptop/mobile chips)

TPU (Tensor Processing Unit): - ❌ Not practical for consumer desktop use - TPUs are Google's proprietary accelerators (Cloud TPU or Google Edge TPU) - Edge TPUs are underpowered for real-time STT of Whisper-scale models - Cloud TPUs are rental-only and prohibitively expensive for continuous STT use - Limited software compatibility with Whisper ecosystem

Special Consideration: If You Must Keep Current AMD GPU

If you're not ready for a full build and want to add a second GPU with your current setup, here's what to consider:

Prerequisites: - Verify your motherboard has a second PCIe x16 slot (or at least x8) - Ensure adequate PCIe lane allocation from CPU - Check power supply capacity (likely need 850W+ for dual-GPU) - Verify case airflow can handle additional heat

Budget Second GPU Options (~\$300-400): - **NVIDIA RTX 3060 12GB** (used market): Good VRAM for STT, reasonable performance - **NVIDIA RTX 4060 8GB** (new): Newer architecture but limited VRAM

Setup Configuration: - AMD GPU: Primary display output - NVIDIA GPU: Dedicated to CUDA compute (STT, AI workloads) - Use `CUDA_VISIBLE_DEVICES` environment variable to explicitly route workloads to NVIDIA GPU - Set display manager to use AMD GPU to avoid NVIDIA driver overhead on display tasks

Practical Implementation for STT Workloads

Regardless of which option you choose, here's how to optimize for STT:

Software Stack: 1. **faster-whisper** (recommended): CTranslate2-based, highly optimized, low VRAM usage - large-v3 model runs well on 8GB VRAM - 2-3x faster than OpenAI's Whisper implementation - Significantly lower memory footprint

1. **whisper.cpp**: If you want CPU fallback option

- Uses CUDA when available
- Excellent quantized model support

2. **Hugging Face Transformers**: If you need fine-tuning capabilities

- More VRAM intensive
- Slower inference than faster-whisper

VRAM Requirements by Whisper Model: | Model Size | Minimum VRAM (faster-whisper) | Recommended VRAM | |-----|-----|-----| | tiny | 1GB | 2GB | | base | 1GB | 2GB | | small | 2GB | 4GB | | medium | 4GB | 6GB | | large-v2/v3| 6GB | 10GB |

Real-Time STT Performance Targets: - For real-time transcription (1x speed or faster), you want 4GB+ VRAM - For comfortable headroom with large-v3 and parallel processing, 12GB+ VRAM is ideal

Timeline Recommendation

Immediate (if needed): - Continue using your AMD GPU with ROCm for STT - Consider **whisper.cpp** with CPU offloading if ROCm is problematic

Short-term (3-6 months): - If STT performance is blocking your workflow, consider a used RTX 3060 12GB as a second GPU stopgap - Only if dual-GPU setup is viable on your current system

Next build/major upgrade (12-24 months): - Replace with single NVIDIA RTX 4070 Ti Super 16GB or equivalent next-gen card - This will serve you better than any dual-GPU configuration

Additional Considerations

Power Efficiency: Modern NVIDIA GPUs have excellent idle power management. If you're running STT intermittently throughout the day (not 24/7), the GPU will mostly idle at 10-30W, spiking only during active transcription.

Future-Proofing: STT models are trending toward larger, more capable architectures (Whisper large-v3, Distil-Whisper, Canary). Having 16GB VRAM provides headroom for these developments.

Local LLM Consideration: If you expand your local LLM usage, 16GB VRAM enables: - 13B parameter models at good speed (Q4 quantization) - 7B parameter models at full precision - Simultaneous STT + small LLM workloads

Summary

Best Path Forward: 1. **Next build:** Single NVIDIA RTX 4070 Ti Super 16GB (or equivalent) 2. **Current system:** Continue with AMD + ROCm or consider budget second NVIDIA GPU only if current performance is blocking work 3. **Hardware type:** GPU only—NPUs and TPUs are not suitable for desktop STT workloads

The single powerful NVIDIA GPU approach provides the best balance of performance, flexibility, power efficiency, and system simplicity for your STT-focused workload.

Generated by Claude Code (Anthropic) - Please validate recommendations against your specific motherboard, PSU, and case specifications before purchasing.

Part VI: AMD GPU Optimization

AMD-specific hardware considerations

Amd Gpu Engines Comparison

Question

With an AMD GPU (rather than NVIDIA), what are the best inference engines for ASR that have good AMD/ROCm support? And when converting models from safe-tensors format after fine-tuning, which formats should you target to work well with AMD GPUs?

Answer

Running speech-to-text (STT) workloads locally on AMD GPUs presents unique challenges compared to NVIDIA's more mature CUDA ecosystem. This guide provides a comprehensive comparison of the most reliable engines for AMD GPU-accelerated STT inference.

Current State of AMD GPU Support for STT

The Challenge

AMD GPU support for AI workloads relies primarily on ROCm (Radeon Open Compute), which has historically lagged behind NVIDIA's CUDA in terms of software support and ecosystem maturity. Many popular inference engines were built with CUDA as the primary target, with AMD support added later or not at all.

Engine Comparison

1. Faster-Whisper (Recommended)

Status: Most reliable option for AMD GPU acceleration

Key Details: - Uses CTranslate2 backend, which has ROCm support - Supports AMD GPUs through ROCm (tested with gfx1100, gfx1101, gfx1030, and other RDNA architectures) - Offers 3-4x faster inference than OpenAI's Whisper while maintaining accuracy - Lower VRAM requirements than original Whisper

Installation:

```
pip install faster-whisper
```

ROCM Requirements: - ROCm 5.4+ recommended - Proper `HSA_OVERRIDE_GFX_VERSION` may be needed for some cards - For RDNA 3 (RX 7000 series): `HSA_OVERRIDE_GFX_VERSION=11.0.0` or `11.0.1`

Verification of GPU Usage:

```
watch -n 1 rocm-smi  
  
watch -n 1 'rocm-smi --showuse --showmeminfo vram'
```

Pros: - Best balance of speed, accuracy, and AMD GPU support - Active development and community - Good documentation for ROCm setup

Cons: - Initial setup can be tricky - ROCm version compatibility issues

2. whisper.cpp

Status: Mixed AMD GPU support - unreliable for production

Key Details: - Primarily CPU-optimized (uses SIMD, AVX, etc.) - HIP (ROCM) support exists but is experimental - Must be compiled from source with specific flags for AMD GPU support - GPU acceleration often doesn't engage properly

Why Your GPU Monitoring Showed No Activity: The app you're using likely: 1. Uses a pre-compiled binary without ROCm support 2. Falls back to CPU when GPU initialization fails 3. Doesn't have proper ROCm runtime environment configured

When to Use: - CPU-only inference (where it excels) - Embedded/edge devices - When you need minimal dependencies

Pros: - Excellent CPU performance - Low memory footprint - Fast for CPU-only workloads

Cons: - AMD GPU support is experimental and unreliable - Requires manual compilation with HIP support - Often falls back to CPU silently

3. OpenAI Whisper (Original)

Status: No direct AMD GPU support through PyTorch

Key Details: - Built on PyTorch with CUDA backend - PyTorch has experimental ROCm support (separate installation) - Slower than optimized alternatives - Higher VRAM requirements

ROCM PyTorch Installation:

```
pip install torch torchvision torchaudio --index-url https://download.pytorch.org/wheel/rocm6.0
```

Pros: - Reference implementation - Most accurate (though Faster-Whisper matches it) - Good for testing/validation

Cons: - Slowest inference - ROCm PyTorch support is hit-or-miss - High VRAM usage - Not optimized for inference

4. Whisper-JAX

Status: Limited AMD support through experimental ROCm JAX builds

Key Details: - Built on JAX framework - JAX has experimental ROCm support but very limited - Primarily designed for TPU/CUDA

When to Use: - You have specific JAX ROCm expertise - Research/experimental workloads

Recommendation: Avoid for AMD GPU production use

5. Whisper.onnx / ONNX Runtime

Status: Growing AMD GPU support through DirectML and ROCm

Key Details: - ONNX Runtime has ROCm execution provider - Can convert Whisper models to ONNX format - DirectML provider works on Windows with AMD GPUs

Installation:

```
pip install onnxruntime-rocml  
  
python -m optimum.exporters.onnx --model openai/whisper-base whisper-onnx/
```

Pros: - Cross-platform - Good optimization potential - Growing ecosystem

Cons: - Requires model conversion - ROCm provider less mature than CUDA - More complex setup

Ranking for AMD GPU Users

Tier 1: Production-Ready

1. **Faster-Whisper** - Best overall choice for AMD GPUs

- Reliability: 
- Performance: 
- Ease of Setup: 

Tier 2: Workable with Caveats

1. **OpenAI Whisper + ROCm PyTorch** - Reference implementation

- Reliability: 
- Performance: 
- Ease of Setup: 

2. **ONNX Runtime (ROCm provider)** - For specific use cases

- Reliability: 
- Performance: 
- Ease of Setup: 

Tier 3: Not Recommended for AMD GPU

1. **whisper.cpp** - CPU-focused, unreliable GPU support

- Reliability (GPU): 
- Performance (CPU): 
- Ease of Setup (GPU): 

2. **Whisper-JAX** - Limited ROCm support

- Reliability: 
- Performance: N/A
- Ease of Setup: 

Practical Recommendations

For Your Use Case

Given that you're using an app with whisper.cpp and not seeing GPU activity, here's what's likely happening:

1. **The app is using CPU-only whisper.cpp** - Most pre-packaged apps don't include ROCm-compiled versions
2. **GPU support is claimed but not functional** - The app may have been tested only with NVIDIA GPUs
3. **Silent fallback to CPU** - whisper.cpp will use CPU if GPU initialization fails

Action Plan

Option A: Switch to Faster-Whisper (Recommended)

```

pip install faster-whisper

python << EOF
from faster_whisper import WhisperModel

model = WhisperModel("base", device="cuda", compute_type="float16")
segments, info = model.transcribe("audio.wav")

for segment in segments:
    print(f"[{segment.start:.2f}s -> {segment.end:.2f}s] {segment.text}")
EOF

watch -n 1 rocm-smi

```

Option B: Verify whisper.cpp ROCm Support If you want to stick with your current app: 1. Check if the app supports custom whisper.cpp builds 2. Compile whisper.cpp with HIP support:

```

git clone https://github.com/ggerganov/whisper.cpp
cd whisper.cpp
mkdir build && cd build
cmake .. -DWHISPER_HIPBLAS=ON
make

```

1. Replace the app's whisper.cpp binary with your ROCm-enabled build

Option C: Use PyTorch ROCm + Original Whisper For research/development:

```

pip install torch torchvision torchaudio --index-url https://
download.pytorch.org/whl/rocm6.0

pip install -U openai-whisper

python -c "import torch; print(torch.cuda.is_available())"

```

Verifying GPU Usage on AMD

ROCM System Management Interface

```
rocm-smi

rocm-smi --showuse --showmeminfo vram --showtemp

watch -n 1 rocm-smi
```

Process-Specific GPU Usage

```
sudo apt install radeontop
radeontop

rocm-smi --showpids
```

PyTorch ROCm Verification

```
import torch
print(f"ROCM available: {torch.cuda.is_available()}")
print(f"ROCM version: {torch.version.hip}")
print(f"Device name: {torch.cuda.get_device_name(0)})")
```

Common Pitfalls

1. Silent CPU Fallback

Many inference engines will silently fall back to CPU if GPU initialization fails. Always verify GPU usage with monitoring tools.

2. HSA_OVERRIDE_GFX_VERSION

RDNA 2/3 GPUs often need:

```
export HSA_OVERRIDE_GFX_VERSION=11.0.0 # For gfx1100 (RX 7900 XTX)
export HSA_OVERRIDE_GFX_VERSION=11.0.1 # For gfx1101 (RX 7800 XT)
export HSA_OVERRIDE_GFX_VERSION=10.3.0 # For gfx1030 (RX 6000 series)
```

3. ROCm Version Compatibility

Different inference engines support different ROCm versions. Check compatibility before installation.

4. Pre-compiled Binaries

Most pre-compiled applications and Python wheels are CUDA-only. AMD GPU support often requires:

- Custom compilation
- Specific ROCm wheels
- Environment configuration

Model Recommendations for AMD GPUs

When using Faster-Whisper or other ROCm-enabled engines:

VRAM Considerations

- **4-6GB VRAM:** `tiny`, `base`, `small` models
- **8-12GB VRAM:** `medium` model
- **16GB+ VRAM:** `large-v2`, `large-v3` models

Quantization

Faster-Whisper supports various quantization levels:

- `float16` - Best quality, 2x memory reduction
- `int8` - Good quality, 4x memory reduction
- `int8_float16` - Hybrid approach (recommended)

```
model = WhisperModel("large-v3", device="cuda", compute_type="int8_float16")
```

Future Outlook

Improving AMD Support

- ROCm 6.x+ shows significant improvements

- More inference frameworks adding ROCm support
- Community-driven optimizations growing

Alternative Approaches

- **Containers:** Pre-configured ROCm containers simplify setup
- **Cloud inference:** Consider cloud GPUs for production if local AMD support remains problematic
- **Hybrid approach:** Use AMD GPU for development, NVIDIA for production

Conclusion

For AMD GPU users running local STT workloads, **Faster-Whisper** is currently the most reliable choice. It offers: - Proven ROCm support - Excellent performance - Active development - Straightforward setup

The whisper.cpp issues you're experiencing are common - the engine excels at CPU inference but has unreliable AMD GPU support. Unless your application specifically requires whisper.cpp, switching to Faster-Whisper will provide better AMD GPU utilization and performance.

This document was generated by Claude Code as part of Daniel Rosehill's STT Fine-Tuning Notebook. While comprehensive, please verify specific technical details and version compatibility for your use case. AMD GPU support in AI workloads evolves rapidly.

Amd Rocm Inference Optimization

Question

With an AMD GPU and ROCm, what inference engines have the best support for ASR, and what model formats should you target when converting from safe-tensors after fine-tuning?

Answer

The AMD/ROCm Reality Check

You're right—NVIDIA's CUDA ecosystem dominates AI, and AMD/ROCm support is more limited. However, for ASR specifically (and Whisper in particular), there are well-supported pathways that work excellently on AMD.

Best-Supported Inference Engines for AMD + ROCm

1. CTranslate2 (Recommended - Best Performance)

Why CTranslate2: - Purpose-built for transformer inference optimization - Excellent ROCm support (officially supported) - Used by Faster-Whisper (the fastest Whisper implementation) - 4x faster than PyTorch, significantly lower memory usage - Supports quantization (INT8, FP16)

ROCm Compatibility: - Works with ROCm 5.0+ - Your GPU (gfx1101 - Navi 32) is well-supported - Requires `HSA_OVERRIDE_GFX_VERSION=11.0.1` (which you're already using)

Target Format:

```
Safetensors/PyTorch → CTranslate2 format (.ctranslate2/)
```

Conversion Process:

```
pip install ctranslate2

ct2-transformers-converter --model /path/to/finetuned-whisper \
    --output_dir /path/to/ctranslate2-model \
    --quantization float16 # or int8 for faster inference
```

Why This Works for AMD: - CTranslate2 uses optimized ROCm kernels - Well-maintained AMD support - Active community using it on AMD GPUs

2. ONNX Runtime with ROCm Execution Provider

Why ONNX Runtime: - Open standard (ONNX format) - Microsoft-backed with official ROCm support - Good performance (though not as fast as CTranslate2 for Whisper) - Wide compatibility across frameworks

ROCm Compatibility: - ONNXRuntime 1.14+ has ROCmExecutionProvider - Works on gfx1101 with ROCm 5.4+

Target Format:

```
Safetensors/PyTorch → ONNX (.onnx)
```

Conversion Process:

```
from optimum.onnxruntime import ORTModelForSpeechSeq2Seq
from transformers import WhisperProcessor

model = ORTModelForSpeechSeq2Seq.from_pretrained(
    "path/to/finetuned-whisper",
    export=True,
    provider="ROCMExecutionProvider"
)

model.save_pretrained("path/to/onnx-model")
```

Optimization:

```
python -m onnxruntime.quantization.preprocess \
--input model.onnx \
--output model-quantized.onnx
```

3. PyTorch with ROCm Backend (Fallback Option)

Why PyTorch: - Native format (no conversion needed) - Most flexible for experimentation - Good ROCm support (AMD maintains torch-rocm) - Easier debugging

ROCm Compatibility: - PyTorch 2.0+ has solid ROCm support - Works directly with safetensors/ PyTorch checkpoints

Target Format:

```
Safetensors/PyTorch (native) - no conversion needed
```

Usage:

```

import torch
from transformers import WhisperForConditionalGeneration

model = WhisperForConditionalGeneration.from_pretrained(
    "path/to/finetuned-whisper"
).to("cuda") # "cuda" works with ROCm

model = torch.compile(model)

```

Performance: - Slower than CTranslate2 or ONNX - Higher memory usage - But most straightforward for debugging

Comparison Table

Engine	Performance	ROCm Support	Conversion Complexity	Best Use Case
CTranslate2	★★★★★	★★★★★	Medium	Production inference
ONNX Runtime	★★★★★	★★★★★	Medium	Cross-platform deployment
PyTorch	★★★★	★★★★★	None	Development/ debugging

Recommended Workflow for Your AMD Setup

Primary Path: CTranslate2 (Faster-Whisper)

This is the optimal choice for your AMD GPU:

```

ct2-transformers-converter \
--model ./finetuned-whisper-medium \
--output_dir ./finetuned-whisper-medium-ct2 \
--quantization float16

pip install faster-whisper

```

```

from faster_whisper import WhisperModel

model = WhisperModel(
    "path/to/finetuned-whisper-medium-ct2",
    device="cuda", # Works with ROCm
    compute_type="float16"
)

segments, info = model.transcribe("audio.wav")
for segment in segments:
    print(f"[{segment.start:.2f}s -> {segment.end:.2f}s] {segment.text}")

```

Why This Works Well on AMD:

1. **Optimized Kernels:** CTranslate2 uses ROCm-optimized kernels
2. **Lower Memory:** Your 7700 XT/7800 XT has less VRAM than NVIDIA equivalents—CTranslate2's efficiency helps
3. **Proven Track Record:** Many AMD users successfully run faster-whisper
4. **Active Maintenance:** CTranslate2 team actively supports ROCm

Format Conversion Summary

Post Fine-Tuning Workflow:

1. Training Output:
 - └ safetensors (raw weights)
 - └ pytorch_model.bin
 - └ config.json
2. Convert to Target Format:
 - └ CTranslate2 (RECOMMENDED for AMD)
 - └ Use ct2-transformers-converter
 - └ ONNX (Good alternative)
 - └ Use optimum.onnxruntime
 - └ Keep PyTorch (Development only)
 - └ No conversion needed

AMD-Specific Optimizations

Environment Variables (You're Likely Already Using):

```
export HSA_OVERRIDE_GFX_VERSION=11.0.1 # For gtx1101
export ROCM_PATH=/opt/rocm
export ROC_ENABLE_PRE_VEGA=0
```

Quantization Strategy: - **FP16**: Best balance (2x faster, minimal quality loss) - **INT8**: 4x faster, slight quality degradation - **FP32**: Slowest, unnecessary for inference

Batch Size Tuning: Your 7700 XT/7800 XT has 12GB VRAM: - Whisper tiny: batch size 16-32 - Whisper small: batch size 8-16 - Whisper medium: batch size 4-8 - Whisper large: batch size 1-2 (or use quantization)

What NOT to Use on AMD

✗ **TensorRT**: NVIDIA-only, won't work ✗ **CUDA-specific libraries**: FlashAttention, etc. ✗ **Some quantization tools**: GPTQ, AWQ (CUDA-focused)

Bottom Line Recommendation

For your AMD GPU (gtx1101) + ROCm setup:

1. **Best Performance**: Fine-tune in PyTorch → Convert to CTranslate2 → Use faster-whisper
2. **Best Compatibility**: ONNX Runtime with ROCm execution provider
3. **Easiest Debugging**: Stay in PyTorch

The conversion command you'll use most:

```
ct2-transformers-converter \
--model /path/to/your-finetuned-whisper \
--output_dir /path/to/optimized-model-ct2 \
--quantization float16
```

This gives you near-NVIDIA performance on AMD hardware for ASR inference.

Note: This guidance was generated by Claude Code (claude-sonnet-4-5) for Daniel Rosehill's STT Fine-Tuning Notebook. ROCm support varies by version—always verify compatibility with your specific ROCm version (`rocm-smi --showdriverversion`). For production deployments, test inference performance with your specific audio data.

Gpu Vram Requirements Whisper

The Observation

Running Whisper Medium on an AMD Radeon RX 7700 XT (8GB VRAM) via whisper.cpp shows:

- GPU usage jumping to **100% during inference**
- Inference happens every few seconds during streaming transcription
- Surprising that Medium model maxes out the GPU

Questions:

1. Does 100% GPU usage mean the model is too large?
2. How much VRAM/GPU power do you really need for quality local STT?
3. Is hitting 90-100% GPU utilization during inference problematic?

Short Answer

100% GPU usage during inference is completely normal and expected—it's actually ideal! This means:

- You're using your GPU efficiently
- The model is running at full speed
- This is NOT a problem or bottleneck
- You do NOT need a bigger GPU

The concern about “maxing out” the GPU is based on a misconception: Unlike gaming where 100% GPU means you're bottlenecked, in AI inference, 100% utilization during processing is the *goal*. Between inference bursts, GPU usage drops back down—this is normal streaming behavior.

Understanding GPU Utilization in AI Inference

Gaming/Graphics Workload (Continuous)

Timeline: [██████████] 100% sustained
Meaning: GPU struggling to keep up with frame rate demands
Problem: You need a better GPU or lower settings

In gaming, continuous 100% means bottleneck.

AI Inference Workload (Bursty)

Timeline: [██████ — ████ — ████ — ████] Bursts to 100%
Meaning: GPU efficiently processing, then idle, then processing
Normal: This is optimal behavior

In AI inference, bursts to 100% mean efficient utilization.

Your Whisper.cpp Streaming Case

Every 3 seconds:

[Recording audio]	GPU: 0-5%	← Waiting for audio
[Inference begins]	GPU: 100%	← Processing audio
[Inference complete]	GPU: 0-5%	← Done, waiting

This pattern is perfect. You want GPU to spike to 100% during the brief inference, then return to idle.

VRAM Requirements vs GPU Compute

Two separate concerns:

1. VRAM Capacity (Memory Size)

What it determines: Which model size you can load

Whisper model VRAM requirements:

Model	Parameters	VRAM (FP16)	VRAM (INT8)	VRAM (Q5)	VRAM (Q4)
Tiny	39M	~150 MB	~80 MB	~50 MB	~40 MB
Base	74M	~290 MB	~150 MB	~100 MB	~80 MB
Small	244M	~950 MB	~480 MB	~350 MB	~280 MB
Medium	769M	~3.0 GB	~1.5 GB	~1.1 GB	~900 MB
Large-v2	1550M	~6.0 GB	~3.0 GB	~2.2 GB	~1.8 GB
Large-v3	1550M	~6.0 GB	~3.0 GB	~2.2 GB	~1.8 GB
Large-v3-turbo	809M	~3.1 GB	~1.6 GB	~1.2 GB	~1.0 GB

Your AMD RX 7700 XT (8GB VRAM) can handle:

- ✓ Medium (FP16, INT8, all quantizations) with room to spare
- ✓ Large-v3-turbo (FP16, INT8, quantized)
- ✓ Large-v2/v3 (INT8 and quantized versions)
- ✗ Large-v2/v3 (FP16) - would use ~6GB, leaving only 2GB for system

Whisper.cpp default: Usually uses Q5 or Q4 quantization, so your 8GB is plenty even for Large models

2. GPU Compute Power (Processing Speed)

What it determines: How *fast* inference runs

AMD RX 7700 XT specs:

- Compute Units: 54
- Peak FP16 performance: ~35 TFLOPS
- Memory bandwidth: 432 GB/s
- Architecture: RDNA 3 (Navi 32)

This is a mid-to-upper-tier GPU—very capable for local AI.

Decoding the “100% GPU Usage”

What’s Actually Happening

When whisper.cpp processes audio:

```
audio_chunk = capture_audio(3_seconds) # GPU: 0%

gpu_buffer = transfer_to_gpu(audio_chunk) # GPU: 5-10%

transcription = model.forward(gpu_buffer) # GPU: 100%
                                         # Duration: 0.5-2 seconds

print(transcription) # GPU: 0%

time.sleep(1) # GPU: 0%
```

Your observation: GPU hits 100% during `model.forward()`

This is correct and optimal! You want the GPU to work at full capacity during inference.

Why This Isn’t a Problem

1. Inference is short: Even at 100%, each inference burst lasts only 0.5-2 seconds

2. Duty cycle is low: If inference takes 1 second every 3 seconds, that’s only 33% average utilization

3. Temperature managed: AMD GPUs throttle if they overheat—100% for 1 second won’t cause thermal issues

4. No frame drops: Unlike gaming, there’s no frame rate to drop. Either inference finishes or it doesn’t—and yours is finishing successfully.

The Display/System Concern

Your question: “Doesn’t GPU also need to run displays?”

Answer: GPU time-shares, and display composition uses negligible compute:

```
GPU time allocation (simplified):  
[Inference: 0.8s] [Display: 0.01s] [Idle: 1.99s] [Inference: 0.8s] ...
```

Display needs: ~5-20ms per frame at 60 FPS = ~0.3-1% of GPU time

Even at 100% inference utilization, there's enough GPU time between frames for display updates. You'd notice display issues (stuttering, lag) if this were a problem—and you haven't mentioned any.

Real-World Performance Expectations

Inference Speed (Real-Time Factor)

Real-Time Factor (RTF): How long to transcribe vs audio duration

```
RTF = inference_time / audio_duration  
  
RTF = 1.0 → Real-time (1 second to process 1 second of audio)  
RTF = 0.5 → 2× real-time (0.5 seconds to process 1 second of audio)  
RTF = 2.0 → 0.5× real-time (2 seconds to process 1 second of audio)
```

Your AMD RX 7700 XT expected performance (whisper.cpp with ROCm):

Model	RTF (approx)	Meaning
Tiny	0.05-0.1	10-20× real-time
Base	0.1-0.15	6-10× real-time
Small	0.2-0.3	3-5× real-time
Medium	0.4-0.6	1.6-2.5× real-time
Large-v3	0.7-1.0	1-1.4× real-time
Large-v3-turbo	0.5-0.7	1.4-2× real-time

Your Medium at ~100% GPU usage likely achieving RTF ≈ 0.5, meaning it's processing 2× faster than real-time—which is *excellent* for streaming transcription.

What “Quality Results” Requires

Myth: High GPU utilization = poor quality results

Reality: Quality depends on:

1. **Model accuracy** (Medium is highly accurate)
2. **Successful completion** (your transcriptions are working)
3. **Reasonable latency** (you’re getting results every few seconds)

GPU utilization percentage is irrelevant to output quality. As long as inference completes successfully (which it is), you’re getting full-quality results.

When Would You Actually Need More GPU?

You’d need a bigger GPU if:

1. Real-Time Factor Too Slow

Your audio: 3 seconds
Inference time: 4+ seconds
Result: Transcription falls behind

Your case: Not happening—Medium is processing faster than real-time

2. Running Multiple Models Simultaneously

Whisper + Stable Diffusion + LLM inference
Result: Out of VRAM or extreme slowdown

Your case: Only running Whisper

3. Batch Processing Many Files

Processing 100 audio files
Want: 5x faster throughput
Result: Larger GPU would help batch processing

Your case: Streaming transcription—batch speed less relevant

4. Using Unquantized Large Models

```
Loading Large-v3 in FP16: 6GB VRAM  
Remaining: 2GB for system  
Result: Might struggle with very large models in full precision
```

Your case: whisper.cpp uses quantization—you’re fine

Optimizing Your Current Setup

You don’t need a new GPU, but you can optimize:

1. Ensure ROCm is Properly Configured

```
rocm-smi
```

If whisper.cpp falls back to CPU: Performance would be much worse, but wouldn’t show 100% GPU usage

2. Try Large-v3-Turbo

```
whisper.cpp --model large-v3-turbo input.wav
```

Expected: GPU still hits 100% during inference (which is fine), but possibly slightly longer bursts

3. Check Thermal Throttling

```
watch -n 1 rocm-smi
```

If throttling: GPU automatically reduces clock speed when hot—this *would* hurt performance, but 100% utilization doesn’t necessarily mean throttling

4. Monitor VRAM Usage, Not Just Utilization

```
rocm-smi | grep "Memory"
```

If VRAM is nearly full (>7GB): Then you're at the limit

If VRAM usage is low (~1-2GB): You have lots of headroom

Model Selection Guide for Your GPU

Your AMD RX 7700 XT (8GB) can comfortably run:

Recommended for Quality + Speed Balance:

1. Large-v3-Turbo (best choice)

- Accuracy: 90-95% of Large-v3
- Speed: ~1.4-2× real-time on your GPU
- VRAM: ~1GB (Q4 quantization)
- **Best overall option**

2. Medium (what you're using)

- Accuracy: Excellent for most use cases
- Speed: ~2-2.5× real-time on your GPU
- VRAM: ~900MB (Q4 quantization)
- **Very solid choice, no need to change unless you want better accuracy**

If You Want Maximum Accuracy:

3. Large-v3 (quantized)

- Accuracy: Best available
- Speed: ~1-1.4× real-time on your GPU
- VRAM: ~2GB (Q4 quantization)
- **Slight latency increase, but still real-time capable**

If You Want Maximum Speed:

4. Small

- Accuracy: Good for clean audio
- Speed: ~3-5× real-time on your GPU
- VRAM: ~300MB
- **Fast, but noticeably less accurate than Medium**

Comparing Your GPU to Others

Your AMD RX 7700 XT ranks:

GPU Class	Example	Whisper Medium RTF	Can Handle Large?
Entry-level	GTX 1650, RX 6500 XT	0.8-1.2	Barely
Mid-range	RTX 3060, RX 6700 XT	0.5-0.7	Yes (quantized)
Your tier	RX 7700 XT, RTX 3070	0.4-0.6	Yes, easily
High-end	RTX 4070 Ti, RX 7900 XT	0.3-0.4	Yes, very fast
Flagship	RTX 4090, RX 7900 XTX	0.15-0.25	Yes, blazing

You're in a very good tier for local STT. A 4090 would be ~2× faster, but you're already faster than real-time, so it wouldn't meaningfully improve user experience.

The Psychology of 100%

Why 100% *feels* wrong:

- Gaming culture: 100% GPU = “maxed out”, need upgrade
- CPU usage: 100% CPU often means system is struggling
- Temperature concerns: High utilization = heat

Why 100% is actually *right* for AI inference:

- You're paying for compute—use it!

- Burst workload: 100% for 1 second every 3 seconds ≠ sustained load
- Efficient resource usage: Idle GPU is wasted GPU during inference
- No quality impact: Model runs full computation regardless

Better metrics to watch:

- ✓ Inference speed (faster than real-time?)
- ✓ VRAM usage (under 7GB?)
- ✓ Temperature (under 85°C?)
- ✓ Transcription latency (acceptable?)
- ✗ GPU utilization percentage (irrelevant for quality)

Recommendations

What You Should Do

1. Keep using Medium—it's working great!

- Your GPU is handling it well
- 100% utilization during inference is optimal
- Results are good quality

2. Optionally try Large-v3-Turbo

```
whisper.cpp --model large-v3-turbo
```

- Test if accuracy improvement is worth slight latency increase
- Your GPU can handle it

3. Monitor VRAM and temperature, not utilization

```
watch -n 1 'rocm-smi | grep -E "Temperature|Memory"'
```

- VRAM <7GB? ✓ You're fine
- Temperature <85°C? ✓ You're fine
- Utilization 100%? ✓ This is correct!

What You Should NOT Do

X Don't upgrade GPU based on 100% utilization

- You're not bottlenecked
- Inference is faster than real-time
- Quality is excellent

X Don't drop to Small/Tiny to “reduce GPU load”

- You'd lose accuracy for no benefit
- GPU sitting at 50% instead of 100% doesn't help anything
- Use the compute power you have!

X Don't worry about “maxing out” GPU

- This isn't gaming
- Burst loads are normal and healthy
- Your GPU is designed for this

Ballpark VRAM Requirements

For quality local STT with Whisper:

Use Case	Recommended	VRAM Needed	GPU Example
Minimum viable	Small	2GB	GTX 1050 Ti
Good experience	Medium	4GB	GTX 1660
Excellent	Medium/Large-Turbo	6-8GB	RX 7700 XT, RTX 3070
Best	Large-v3	8-12GB	RTX 3080, RX 7900 XT
Overkill	Large + fine-tunes	16GB+	RTX 4090

You're in the “Excellent” tier. You don't need more VRAM for quality local STT—you already have it.

Conclusion

Your AMD RX 7700 XT (8GB VRAM) is more than sufficient for quality local STT.

100% GPU utilization during inference is:

- ✓ Normal and expected
- ✓ Sign of efficient resource usage
- ✓ Not a problem or bottleneck
- ✓ Not affecting quality

You can comfortably run:

- Medium (what you're using)—excellent choice
- Large-v3-Turbo—better accuracy, still good speed
- Large-v3 (quantized)—best accuracy, acceptable speed

You should NOT:

- Worry about 100% GPU spikes
- Think you need to upgrade
- Drop to smaller models to “reduce load”

The display/system concern is unfounded: Display composition takes <1% of GPU time and doesn't compete meaningfully with inference bursts.

Your whisper.cpp setup is working optimally. Enjoy your fast, accurate local transcription!

Note: This document was generated by Claude Code, an AI assistant. Please validate technical details and test recommendations in your specific environment before implementing.