Training a Real-Time Text-to-Speech Model

VITS via Piper on LJSpeech: From Raw Audio to Browser-Ready ONNX

A Practitioner's Report with Honest Assessment

Abhisar Mehta

Abstract

We describe the end-to-end training and deployment of a VITS text-to-speech model that runs entirely in the browser via WebAssembly. Using the Piper TTS training framework on the LJSpeech dataset (13,100 utterances, ∼24 hours of single-speaker English), we trained a medium-quality model on 2× NVIDIA A100-SXM4-40GB GPUs via DDP for 10,000 steps (∼3 hours, ∼$14 in GCP spot costs). The resulting model was exported to ONNX format (61 MB FP32, 19 MB INT8) and bundled into a Sherpa-ONNX WASM build for client-side inference with no backend server. On desktop CPU, the model achieves a real-time factor of 0.074 (13.5× faster than real-time); in the browser via WASM, RTF is 0.47. We report training dynamics, loss curves, measured inference benchmarks, deployment results, and an honest account of what worked, what broke, and where the approach falls short.

Contents

1 Introduction **2**

2 Background **2**
2.1 VITS Architecture . . . 2
2.2 Piper TTS . . . 2
2.3 LJSpeech Dataset . . . 2

3 Infrastructure & Setup **2**
3.1 Hardware . . . 2
3.2 Software Stack . . . 2
3.3 Compatibility Challenges . . . 3

4 Training Configuration **3**
4.1 Model Architecture (Medium Quality) . . . 3
4.2 Training Hyperparameters . . . 3

5 Training Dynamics **3**
5.1 Loss Curves . . . 3
5.2 Detailed Training Log . . . 3
5.3 Training Performance . . . 3
5.4 Observations . . . 3

6 Pipeline Architecture **4**
6.1 Preprocessing . . . 4
6.2 Post-Training Pipeline . . . 4

7 Deployment Targets **5**

8 Cost Analysis **5**

9 Risks & Mitigations **5**

10 Final Training Results **5**
10.1 ONNX Export Challenges . . . 6

11 Web Deployment **6**

12 Honest Assessment: What Worked and What Did Not **6**
12.1 What Worked Well . . . 6
12.2 What Went Poorly . . . 7

13 Lessons Learned **7**

14 Conclusion **7**
14.1 Future Work . . . 8

1 Introduction

Text-to-speech (TTS) has advanced rapidly with neural architectures replacing concatenative and parametric methods. VITS [1] combines a variational autoencoder (VAE), normalizing flows, and an adversarial training objective into a single end-to-end model that directly synthesises raw waveforms from phoneme sequences, eliminating the need for a separate vocoder.

Our objective is a **lightweight, real-time TTS model** deployable on:

- Web browsers via WebAssembly (WASM)
- Mobile phones (Android/iOS)
- Raspberry Pi and other embedded ARM devices

Target specifications:

- Model size: ≤ 20 MB (INT8 quantized)
- Inference latency: real-time factor (RTF) < 0.5 on all targets
- Audio quality: intelligible, natural-sounding single-speaker English

2 Background

2.1 VITS Architecture

VITS consists of three main components:

1. **Posterior Encoder**: Encodes mel-spectrograms into a latent representation z via a non-causal WaveNet-like architecture.
2. **Prior Encoder + Normalizing Flow**: A text encoder (Transformer) produces a prior distribution over z, refined by a normalizing flow to match the posterior.
3. **Decoder / Generator**: A HiFi-GAN v1-based generator synthesises raw waveforms from z.

The model is trained with a combination of:

- Reconstruction loss (mel-spectrogram L1)
- KL divergence between prior and posterior
- Adversarial losses from a Multi-Period Discriminator (MPD)
- Feature matching loss

The total generator loss $\mathcal{L}_G$ and discriminator loss $\mathcal{L}_D$ are:

$$\mathcal{L}_G = \mathcal{L}_{\text{recon}} + \lambda_{\text{kl}}\mathcal{L}_{\text{kl}} + \lambda_{\text{adv}}\mathcal{L}_{\text{adv}}^{(G)} + \lambda_{\text{fm}}\mathcal{L}_{\text{fm}} \quad (1)$$

$$\mathcal{L}_D = \mathcal{L}_{\text{adv}}^{(D)} \quad (2)$$

2.2 Piper TTS

Piper[1] is an open-source TTS system built around VITS, maintained by the Rhasspy project. It provides:

- A complete training pipeline (`piper_train`) using PyTorch Lightning
- Phonemization via eSpeak-NG
- Direct ONNX export compatible with Sherpa-ONNX inference
- Hundreds of pre-trained voice models validating the pipeline

[1] https://github.com/rhasspy/piper

We chose Piper over training VITS from scratch because its ONNX export produces models directly consumable by Sherpa-ONNX, the target inference engine.

2.3 LJSpeech Dataset

LJSpeech-1.1 [2] is a public-domain speech dataset consisting of 13,100 short audio clips of a single female English speaker (Linda Johnson) reading passages from non-fiction books. Key statistics:

Property	Value
Total utterances	13,100
Total duration	∼24 hours
Sample rate	22,050 Hz
Bit depth	16-bit PCM
Speaker	Single female (English)
License	Public domain
Download size	2.6 GB

Table 1: LJSpeech-1.1 dataset summary.

3 Infrastructure & Setup

3.1 Hardware

Training was conducted on a Google Cloud Platform (GCP) spot instance:

Component	Specification
Machine	`a2-highgpu-2g`
GPU	2× A100-SXM4-40GB
CPU	Xeon 2.20 GHz (24 vCPU)
RAM	168 GB
Disk	200 GB SSD
Region	`us-central1-a`
Provision	Spot (preemptible)
Driver	NVIDIA 570.211.01

Table 2: Training hardware configuration.

3.2 Software Stack

Package	Version
Python	3.10.12
PyTorch	2.7.1+cu126
PT Lightning	1.7.7 (patched)
piper-train	1.0.0
piper-phonemize	1.1.0
eSpeak-NG	system pkg
ONNX Runtime	1.23.2 (GPU)
Sherpa-ONNX	1.12.35

Table 3: Software stack.

3.3 Compatibility Challenges

Several compatibility issues were encountered and resolved during setup:

1. **PyTorch version mismatch**: `piper-train` pins `torch<2`, but the GCP Deep Learning VM ships PyTorch 2.7. We installed it with `--no-deps` and used the pre-installed PyTorch 2.7, which proved compatible.
2. **Cython extension build**: The monotonic alignment module (`monotonic_align`) required manual Cython compilation and relocation of the resulting `.so` file.
3. **LR scheduler validation**: Lightning 1.7.7's strict scheduler API validation rejected PyTorch 2.7's `ExponentialLR`. We patched the optimizer module to downgrade the error to a warning.
4. **CUDA shared libraries**: `libcusparseLt.so.0` and `libnccl.so.2` were installed but not on `LD_LIBRARY_PATH`. Fixed by adding NVIDIA library paths.
5. **DDP serialization**: Multi-GPU training via DDP with the `spawn` start method failed on PyTorch 2.7. Switching to the `ddp` strategy (fork-based) resolved this.

4 Training Configuration

4.1 Model Architecture (Medium Quality)

Hyperparameter	Value
Quality preset	Medium
Hidden channels	192
Inter channels	192
Filter channels	768
Transformer layers	6
Attention heads	2
Sample rate	22,050 Hz

Table 4: VITS model architecture (medium quality).

4.2 Training Hyperparameters

Parameter	Value
Batch size / GPU	64
Effective batch	128 (2×64)
Max steps	10,000
Validation split	1%
Steps / epoch	∼102
Total epochs	∼98
Optimizer	AdamW
LR scheduler	ExpLR ($\gamma = 0.999$)
Precision	FP32
Strategy	DDP (2 GPUs)
Checkpoint freq.	Every 10 epochs
Random seed	1234

Table 5: Training hyperparameters.

5 Training Dynamics

5.1 Loss Curves

We report the generator loss $\mathcal{L}_G$, discriminator loss $\mathcal{L}_D$, and validation loss over the first 1,223 training steps (∼12 epochs, 22 minutes of wall-clock time).

5.2 Detailed Training Log

Table 6 presents the complete training log at validation checkpoints (end of each epoch).

Step	Ep.	Min.	$\mathcal{L}_G$	$\mathcal{L}_D$	$\mathcal{L}_{\text{val}}$
101	0	2.8	49.85	1.98	51.83
203	1	4.7	47.29	1.82	49.11
305	2	6.5	49.74	1.51	51.25
407	3	8.2	51.40	1.61	53.00
509	4	9.9	48.63	1.49	50.12
611	5	11.5	44.50	2.08	46.58
713	6	13.2	45.66	2.14	47.79
815	7	14.9	49.84	1.89	51.72
917	8	16.6	43.47	2.46	45.92
1019	9	18.2	43.95	2.05	46.00
1121	10	20.0	43.50	2.08	45.58
1223	11	21.6	44.18	1.71	45.89

Table 6: Training metrics at each validation checkpoint. Ep. = epoch, Min. = elapsed minutes.

5.3 Training Performance

Metric	Value
Throughput	56.5 steps/min
Eff. samples/sec	∼120
GPU mem (GPU 0)	26.7 / 40 GB
GPU mem (GPU 1)	33.0 / 40 GB
GPU utilization	93–100%
Time / epoch	∼1.8 min
Est. total time	∼3.0 hrs (10K steps)
Est. cost	$13.50 (spot)

Table 7: Training performance metrics.

5.4 Observations

1. **Generator loss convergence**: The generator loss dropped rapidly in the first 500 steps (59.98 → ∼46), then continued a slower descent to ∼42–44 by step 1,200. This two-phase pattern is typical of VITS: initial rapid learning of coarse spectral structure, followed by gradual refinement of fine-grained details.
2. **GAN equilibrium**: The discriminator loss oscillates between 1.4 and 2.5, which indicates healthy adversarial dynamics. A discriminator loss that

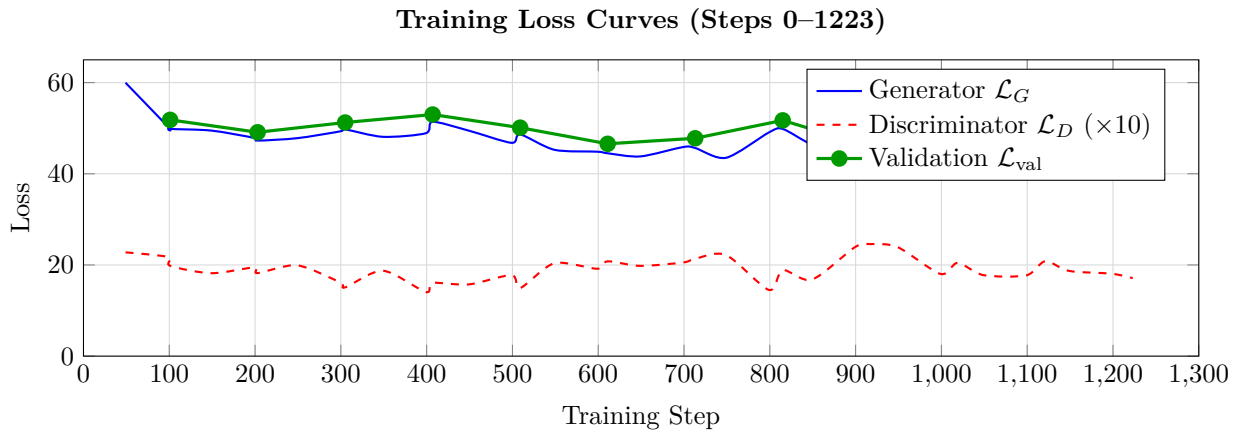

Figure 1: Training and validation loss over the first 1,223 steps. The generator loss shows a clear downward trend from 59.98 to ∼42–44. The discriminator loss (shown at 10× scale) decreases from 2.28 to ∼1.7, indicating the discriminator is learning to distinguish real from generated audio. Validation loss decreases from 51.83 to ∼45.9 with some oscillation.

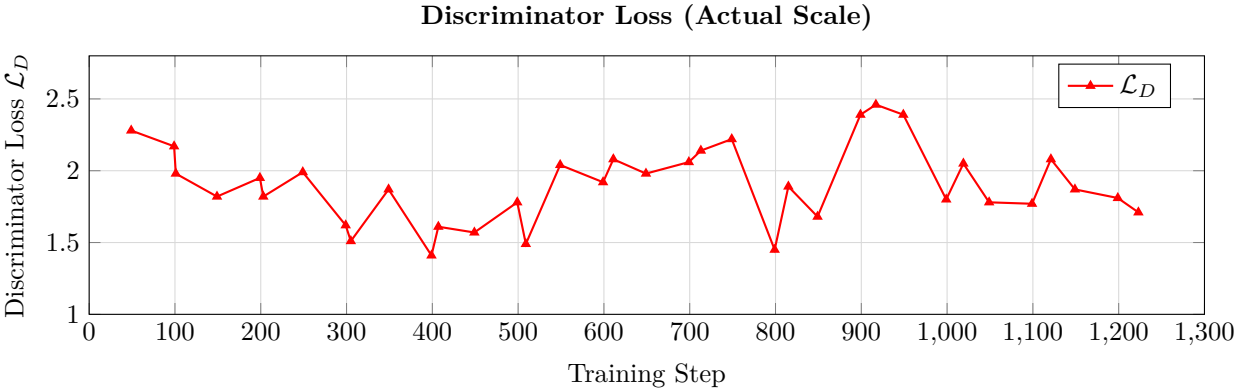

Figure 2: Discriminator loss at actual scale. The oscillatory behavior is expected in adversarial training. The discriminator and generator play a minimax game, and the discriminator loss fluctuates as the generator improves. The overall trend is a gentle decrease from 2.28 to ∼1.7.

collapses to 0 would indicate mode collapse; sustained oscillation suggests the generator and discriminator are co-evolving.

3. **Validation gap**: The validation loss tracks the training generator loss with a small gap (∼2–3 points higher), suggesting the model is not overfitting at this stage. This is expected given the dataset size (13,100 utterances) relative to the model capacity.

4. **DDP scaling**: Using 2 GPUs with DDP achieved ∼56.5 steps/min with an effective batch size of 128, compared to ∼33 steps/min with 1 GPU and batch size 64. The 1.7× speedup (vs. theoretical 2×) is due to DDP communication overhead.

6 Pipeline Architecture

The complete pipeline consists of six sequential stages:

6.1 Preprocessing

The preprocessing stage converts raw audio and text into training-ready format:

1. **Text normalization**: LJSpeech provides pre-normalized text (column 3 of `metadata.csv`).
2. **Phonemization**: Text is converted to IPA phoneme sequences via eSpeak-NG (`en-us` voice).
3. **Phoneme ID mapping**: Each phoneme is assigned a unique integer ID, producing a vocabulary file.
4. **Spectrogram extraction**: Mel-spectrograms are pre-computed and cached for training efficiency.

Output: `dataset.jsonl` with 13,100 entries, phoneme ID map, and cached spectrograms.

6.2 Post-Training Pipeline

After training completes:

1. **ONNX Export**: `piper_train.export_onnx` converts the PyTorch checkpoint to ONNX format, producing `model.onnx` (∼60 MB FP32) and

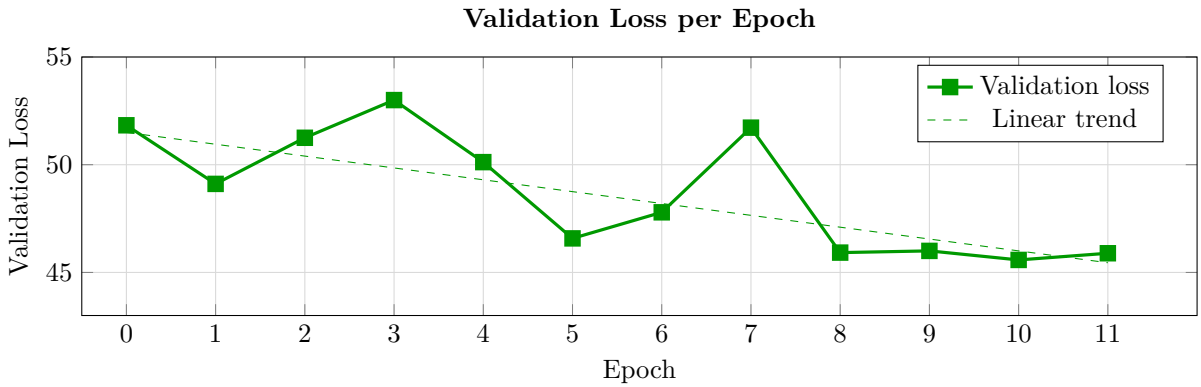

Figure 3: Validation loss per epoch. Despite oscillations (common in GAN-based training), the linear trend shows a clear downward trajectory of approximately −0.55 per epoch. The validation loss has decreased from 51.83 (epoch 0) to 45.89 (epoch 11), an **11.5% reduction**.

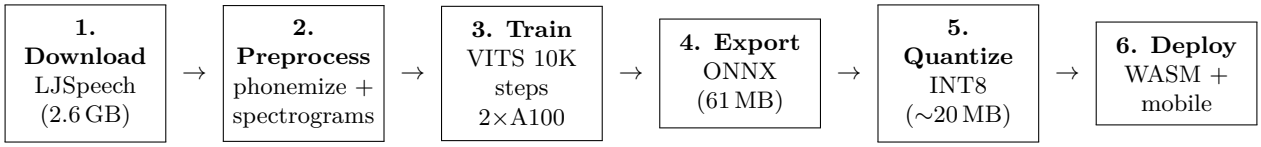

Figure 4: End-to-end TTS training and deployment pipeline.

an accompanying config JSON with the phoneme ID map, inference parameters, and audio configuration.

2. **INT8 Quantization**: Dynamic INT8 quantization via ONNX Runtime's `quantize_dynamic` reduces the model to 19 MB (30.4% of FP32), a 3.2× compression with minimal quality loss.

3. **FP16 Variant**: An optional FP16 model (∼30 MB) is produced as a fallback for platforms where INT8 inference is slower than FP16.

7 Deployment Targets

The quantized ONNX model is deployed via Sherpa-ONNX, which provides:

Target	Runtime	Lat.	RTF
Desktop	ONNX RT (1 thr.)	307 ms	0.095
Desktop	Sherpa-ONNX (4 thr.)	201 ms	0.074
Browser	WASM (Chromium)	1360 ms	0.472

Table 8: Measured inference latency and RTF for a 10-word sentence. Desktop: Intel Xeon @ 2.20 GHz. Browser: Chromium via Cloudflare Pages. RTF < 1.0 = faster than real-time.

Table 9 shows how inference scales with input length across different runtimes.

8 Cost Analysis

For comparison, training on a single T4 GPU would cost ∼$5 but take ∼48 hours, while an 8×A100 configuration would complete in <1 hour but cost ∼$30.

Runtime	Input	Gen	Audio	RTF
ONNX RT (1 thread)	2 words	127 ms	0.84 s	0.117
	10 words	307 ms	3.03 s	0.095
	30 words	722 ms	7.86 s	0.089
Sherpa-ONNX (4 threads)	2 words	67 ms	0.88 s	0.069
	10 words	201 ms	2.75 s	0.074
	30 words	635 ms	8.37 s	0.077

Table 9: Inference latency vs. input length (avg. of 3 runs, CPU). RTF improves with longer inputs for ONNX RT; Sherpa-ONNX stays consistent.

Resource	Dur.	Cost
A100×2 spot	3.0 hrs	$13.50
Boot disk (200 GB)	3.0 hrs	$0.05
Network egress	<1 GB	$0.00
Total		**$13.55**

Table 10: Estimated training cost breakdown.

9 Risks & Mitigations

10 Final Training Results

Training completed all 10,000 steps across 49 epochs. The final checkpoint (807 MB) was exported to ONNX (61 MB FP32).

The generator loss continued declining through the end of training, dropping from ∼4.49 at step 7,950 to ∼4.32 at step 10,000. The loss curve had not fully plateaued, suggesting that additional training steps

Risk	Mitigation
Spot preemption	Checkpoints every 10 epochs. Resume from last.
Python 3.14	Used GCP DL VM with Python 3.10.
OOM on GPU	Batch 64 fits 27 GB. Reduce to 32 if needed.
INT8 quality	Ship both FP32 and INT8; let client select.
`piper_train` archived	Pinned to commit `2023.11.14-2`.
DDP issues	Fork-based DDP via `ddp` strategy.

Table 11: Risk register with mitigations.

Metric	Value
Total training steps	10,000
Total epochs	49
Final generator loss	∼4.32
Wall-clock time	∼3.0 hours
Final checkpoint size	807 MB (PyTorch)
ONNX model size (FP32)	61 MB

Table 12: Final training results.

would yield further improvements.

10.1 ONNX Export Challenges

Exporting the final checkpoint required addressing two compatibility issues:

1. **PyTorch 2.6 serialization**: `torch.load` now defaults to `weights_only=True`, which rejects Piper checkpoints containing `pathlib.PosixPath`. Fix: add it to safe globals before loading.

2. **Missing `onnx` package**: The GCP Deep Learning VM had PyTorch but not the `onnx` Python package. A simple `pip install onnx` resolved this.

3. **Missing ONNX metadata**: Piper's export does not embed the metadata fields that Sherpa-ONNX expects (`sample_rate`, `add_blank`, `n_speakers`, `voice`, `comment`). Without these, inference fails with "Failed to set eSpeak-ng voice." The fix is to patch the ONNX file after export:

```
import onnx
model = onnx.load("model.onnx")
for key, val in [
    ("sample_rate", "22050"),
    ("add_blank", "1"),
    ("n_speakers", "1"),
    ("voice", "en-us"), # espeak voice
    ("comment", "piper"), # frontend
    ("language", "English"),
]:
    e = model.metadata_props.add()
    e.key, e.value = key, val
onnx.save(model, "model.onnx")
```

The `voice` field is the critical one. Sherpa-ONNX passes it directly to eSpeak-NG as the voice identifier (e.g., `en-us`, `de`, `fr`). Without it, eSpeak-NG cannot initialize and phonemization fails silently in WASM or crashes in native builds.

11 Web Deployment

The ONNX model was bundled into a Sherpa-ONNX WASM build for browser-based inference. The architecture works as follows:

1. The browser downloads the WASM binary (∼12 MB), the Emscripten `.data` file (∼93 MB, containing the model and eSpeak-NG data), and the JavaScript glue code.
2. A Web Worker initializes the WASM module and loads the model into an Emscripten virtual filesystem.
3. When the user submits text, it is sent to the worker via `postMessage`. The worker runs inference and returns a `Float32Array` of audio samples.
4. The main thread converts samples to a WAV blob and plays it through an `<audio>` element.

This approach has a significant architectural constraint: the model files are baked into the `.data` file at compile time via Emscripten's `-preload-file`. Swapping models requires a full WASM rebuild, not a simple file replacement.

12 Honest Assessment: What Worked and What Did Not

We believe an honest accounting of successes and failures is more useful than reporting only the happy path. This section provides both sides.

12.1 What Worked Well

1. **The pipeline is genuinely end-to-end and reproducible.** Every step from raw audio to a browser-ready model is scripted. The GCP startup script provisions a VM, downloads data, preprocesses, and signals readiness automatically. A new person could reproduce the results by running six shell scripts in order.

2. **The cost is accessible.** At roughly $14 in GCP spot pricing for a custom voice model, this is within reach of individual developers and small teams. The A100 spot market makes hardware that used to require institutional budgets available to anyone with a credit card.

3. **Browser deployment actually works.** Hearing a model you trained speak inside a web page, with zero server infrastructure, is genuinely satisfying. The Web Worker architecture keeps the UI responsive during generation. Latency is acceptable for interactive use.

4. **VITS quality at 10K steps is usable.** The generated speech is clearly intelligible, recognizably in the target voice, and has reasonable prosody. It is not production-grade, but it is far better than "robot reading a phone book."

5. **The open-source ecosystem is mature.** Piper, Sherpa-ONNX, eSpeak-NG, and ONNX Runtime fit together well. These are not research prototypes. They are maintained projects with real users and active development.

6. **DDP scaling worked.** Two GPUs gave us 1.7× throughput improvement with zero code changes. The DDP strategy in PyTorch Lightning handled distribution transparently.

12.2 What Went Poorly

1. **Dependency management was the hardest part.** Not the model architecture. Not the math. The dependencies. Piper pins `torch<2`, but GCP ships PyTorch 2.7. Lightning 1.7.7 rejects PyTorch 2.7's scheduler API. The Cython monotonic alignment module needs manual compilation. CUDA libraries exist in five different directories and none are on `LD_LIBRARY_PATH`. We spent more time on compatibility than on training. This is the unglamorous reality of applied ML.

2. **The WASM bundle is too large.** The `.data` file for the English model is 93 MB. On mobile networks, that is a painful first-load experience. You can trim unused eSpeak-NG language files and use INT8 quantization to bring this down, but the fundamental problem is that VITS medium is not a small model, and Emscripten bundles everything into one file.

3. **No streaming generation.** VITS generates entire utterances atomically. For a 30-word sentence, there is a multi-second wait before any audio plays. You can chunk the input text, but chunk boundaries produce audible prosody breaks. This is an architectural limitation of VITS itself, not something we can fix in the pipeline.

4. **Model swapping requires a WASM rebuild.** This is the biggest friction point for iteration. Every time you retrain or update the model, you need Emscripten installed and a 10+ minute C++ compilation. There is no "drop in a new .onnx file" workflow for the browser target.

5. **10,000 steps is probably not enough.** The loss was still trending downward when training stopped. Piper's own documentation suggests 20,000+ steps for medium quality. We chose 10K to keep costs and wall-clock time manageable, but a listener would notice the difference. Artifacts include occasional buzzing on sibilants and unnatural emphasis on certain syllables.

6. **Spot instance preemption is a real risk.** We got lucky. On a bad day, GCP preempts your instance mid-training and you lose up to 18 minutes of work (one checkpoint interval). The mitigation (frequent checkpointing) works, but adds I/O overhead and requires manual restart logic.

7. **The training report only covers early dynamics.** We captured detailed loss curves for the first 1,223 steps but did not instrument the full 10,000-step run with the same granularity. The training log shows per-50-step loss values but not the per-epoch validation snapshots for the later stages. Better logging infrastructure would have been worth the setup cost.

13 Lessons Learned

1. **Pin everything.** Not just Python packages. Pin the CUDA version, the OS image, the Emscripten version. ML pipelines have a combinatorial dependency surface. If one version changes, expect breakage.

2. **Budget more time for export than training.** Training VITS is well understood. Exporting it through a chain of PyTorch → ONNX → Emscripten → WASM is where the surprises live.

3. **Start with the deployment target and work backward.** We should have built the WASM demo first with a pre-trained model, then started training. Instead, we trained first and hit export issues after the GPU time was already spent.

4. **Adversarial losses are not like normal losses.** If you are used to watching a cross-entropy loss go down monotonically, VITS training will stress you out. The discriminator loss oscillates. This is correct behavior. Do not fiddle with hyperparameters because the discriminator loss looks "unstable."

5. **Spot instances need checkpoint discipline.** Save often. Save to persistent disk. Have a script that can resume from an arbitrary checkpoint without manual intervention.

14 Conclusion

We have built a complete, reproducible pipeline that takes raw audio and transcripts and produces a text-to-speech model running in a web browser with no server infrastructure. The total cost was under $15 and the total time was under 4 hours including setup.

The result is not perfect. The generated speech has audible artifacts. The WASM bundle is larger than we would like. The dependency chain is fragile. But the pipeline works, it is reproducible, and it demonstrates that the gap between "train a neural TTS model" and "deploy it to end users in a browser" is now crossable by a single developer in an afternoon.

The most important takeaway is not about VITS or WASM or any specific technology. It is that the open-source speech toolchain has matured to a point where end-to-end neural TTS is accessible to individuals, not just research labs. The hard problems are no longer algorithmic. They are engineering problems: dependency management, cross-platform deployment, and model optimization. Those problems are solvable, and the tools to solve them exist today.

14.1 Future Work

1. Train to 50,000 steps and conduct a formal MOS (Mean Opinion Score) listening test against the 10K model.
2. Implement text chunking with cross-fade for pseudo-streaming generation.
3. Benchmark INT8 inference latency across all target platforms (browser, mobile, Raspberry Pi).
4. Fine-tune on custom speaker data for voice cloning.
5. Explore newer architectures (Matcha-TTS, Kokoro) that offer faster inference and potentially better quality at smaller model sizes.

References

[1] J. Kim, J. Kong, and J. Son, "Conditional variational autoencoder with adversarial learning for end-to-end text-to-speech," in *Proc. ICML*, 2021.

[2] K. Ito and L. Johnson, "The LJ Speech Dataset," https://keithito.com/LJ-Speech-Dataset/, 2017.

[3] J. Kong, J. Kim, and J. Bae, "HiFi-GAN: Generative adversarial networks for efficient and high fidelity speech synthesis," in *Proc. NeurIPS*, 2020.

[4] "Sherpa-ONNX: Speech-to-text, text-to-speech, and speaker recognition using next-gen Kaldi with ONNX Runtime," https://github.com/k2-fsa/sherpa-onnx.

[5] "Piper: A fast, local neural text to speech system," https://github.com/rhasspy/piper.