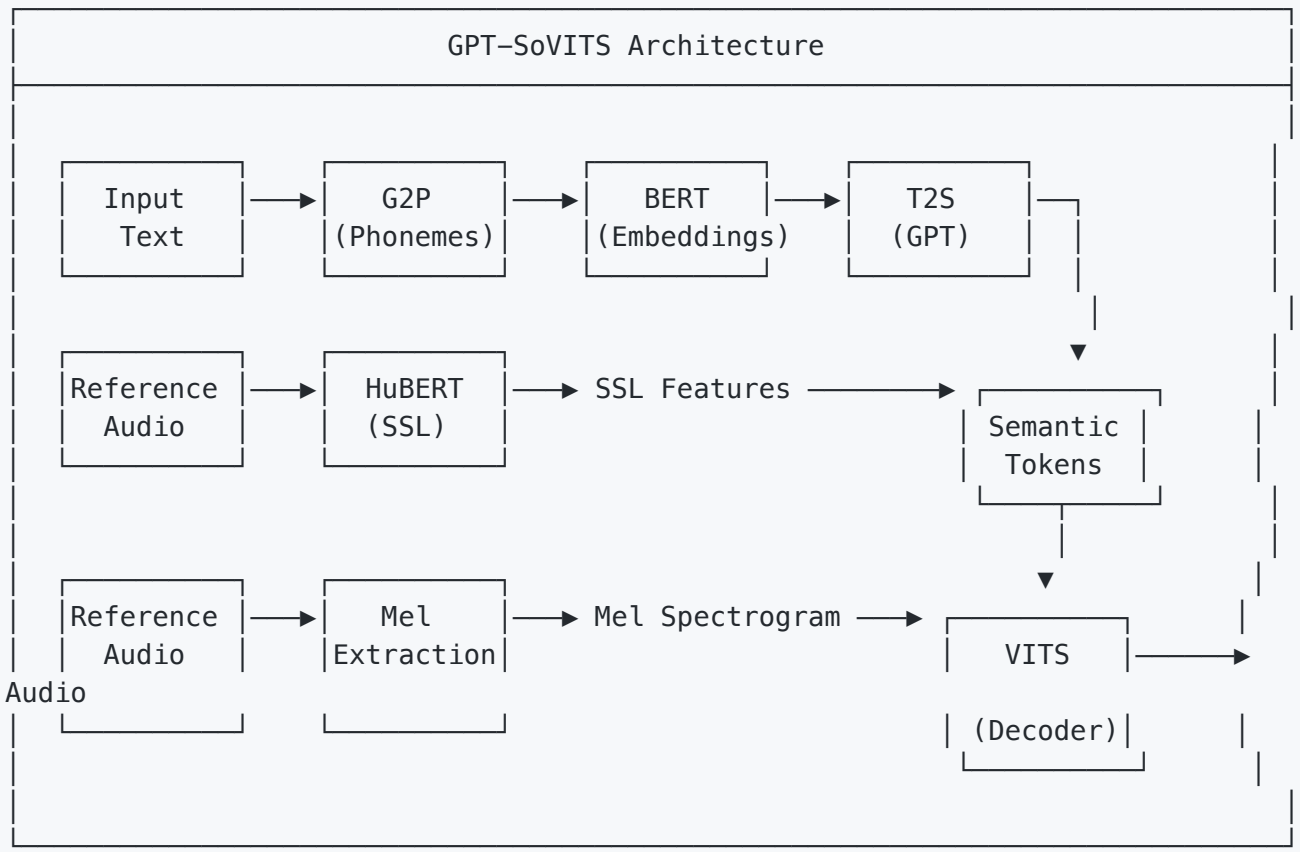
GPT-SoVITS Architecture and Implementation Guide

Table of Contents

Architecture Overview

GPT-SoVITS is a zero-shot/few-shot voice cloning TTS system that combines:

- **GPT-style autoregressive model (T2S)**: Converts text + reference audio into semantic tokens
- **VITS decoder**: Converts semantic tokens into audio waveforms



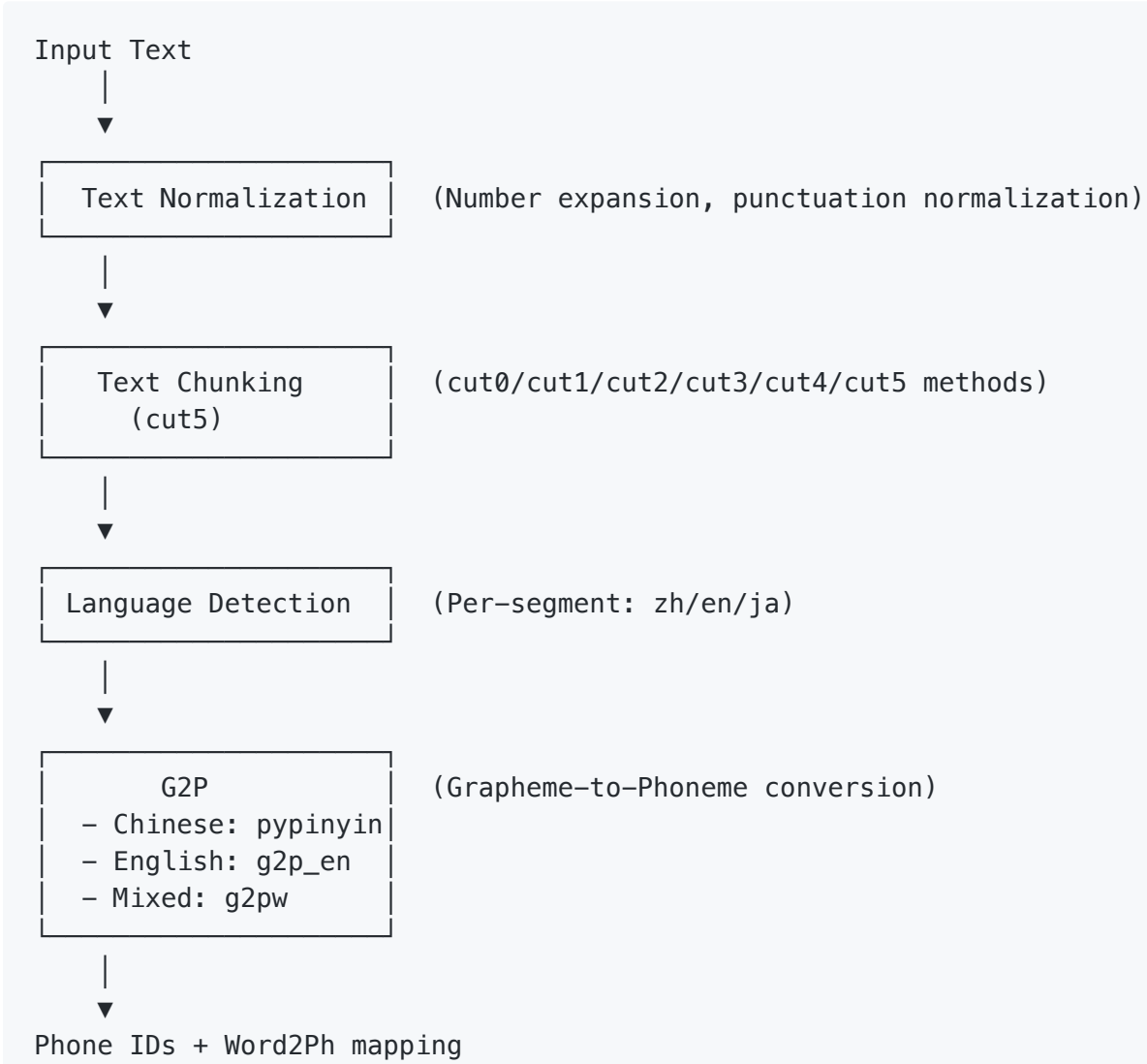
Key Models

Model	Purpose	Input	Output
Chinese-RoBERTa	Text semantic encoding	Phone sequences	1024-dim embeddings
Chinese-HuBERT	Audio SSL features	Reference audio	SSL codes for prompt
T2S (GPT)	Semantic token generation	BERT emb + SSL codes	Semantic token sequence
VITS	Audio synthesis	Semantic tokens + phones + mel	Audio waveform

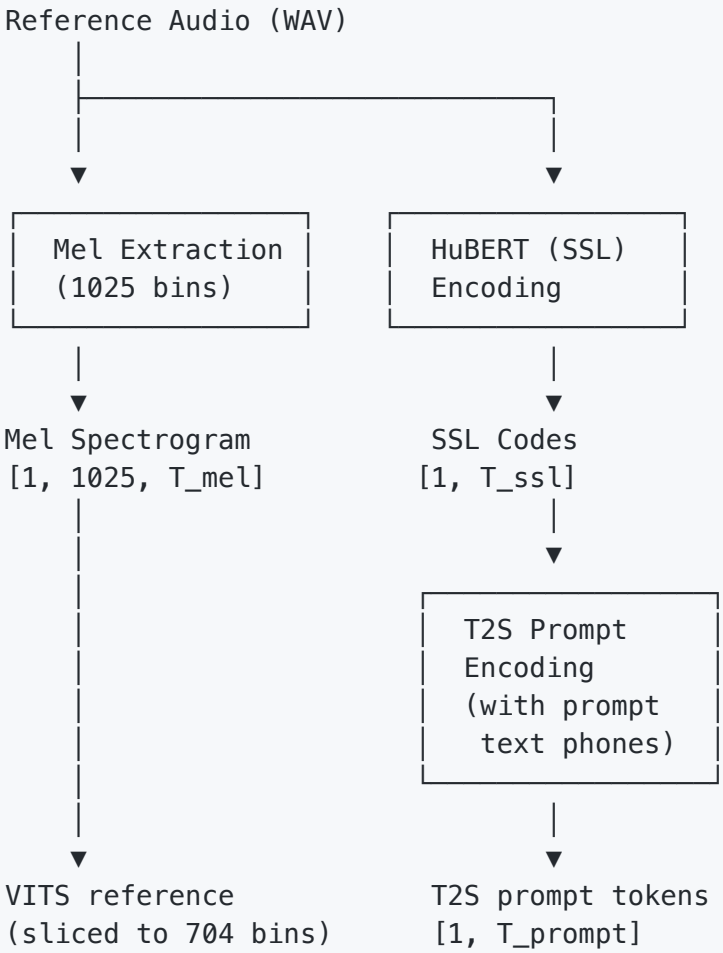
---

End-to-End Pipeline

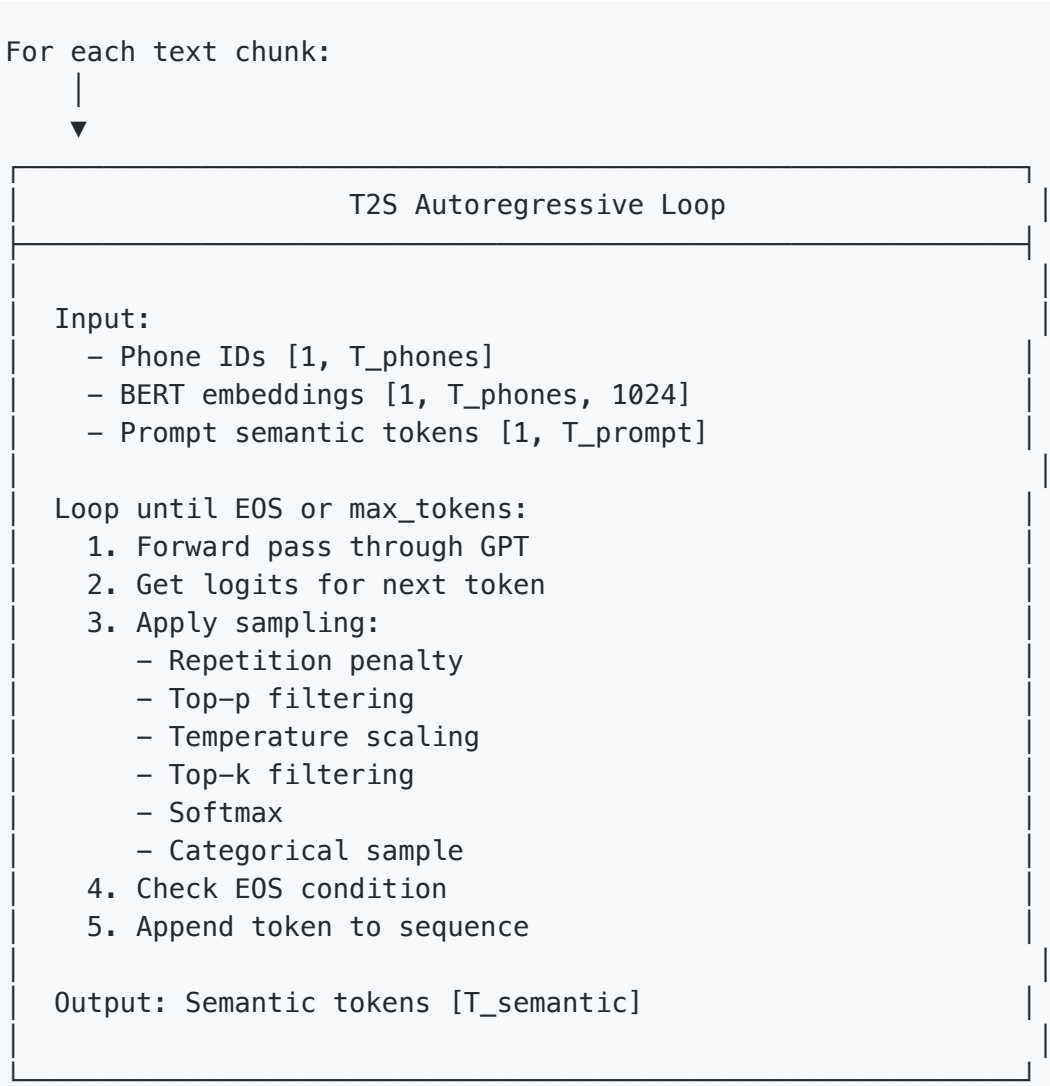
Phase 1: Text Preprocessing



Phase 2: Reference Audio Processing



Phase 3: T2S (Text-to-Semantic) Inference



Phase 4: VITS Synthesis

```
┌──────────────────────────────────────────────────────────────────┐
│                        VITS Decoder                              │
├──────────────────────────────────────────────────────────────────┤
│                                                                  │
│  Two paths based on speed_factor:                                │
│                                                                  │
│  speed == 1.0 (Batched):                                         │
│    - Concatenate ALL chunks' semantic tokens                     │
│    - Concatenate ALL chunks' phone IDs                           │
│    - Single VITS forward pass                                    │
│    - Split output by chunk boundaries                            │
│                                                                  │
│  speed != 1.0 (Per-chunk):                                       │
│    - Process each chunk independently                            │
│    - Apply duration modification                                 │
│                                                                  │
│  Input:                                                          │
│    - codes: Semantic tokens [1, 1, T_codes]                      │
│    - text: Phone IDs [1, T_phones]                               │
│    - refer: Reference mel [1, 704, T_mel]                        │
│                                                                  │
│  Output: Audio waveform [1, 1, W]                                │
│                                                                  │
└──────────────────────────────────────────────────────────────────┘
```

Phase 5: Audio Postprocessing

```
Raw VITS output
    │
    ▼
┌──────────────────────┐
│  Per-chunk processing│
│  - Clip to [-1, 1]   │
│  - Append 0.3s silence│
└──────────────────────┘
    │
    ▼
┌──────────────────────┐
│   Concatenate all    │
│   chunk audio        │
└──────────────────────┘
    │
    ▼
Final audio output (32kHz)
```

Component Deep Dive

1. Text Chunking (cut5)

The `cut5` method splits text at every punctuation mark, then merges short segments.

Punctuation split points:

```
Chinese: 。？！，、；：""''【】『』「」〔〕〈〉《》
English: .?!,;:
Common: \n
```

Merge logic (critical for quality):

```
# Python (correct):
if len(acc) < 5:  # character count
    acc += seg
else:
    yield acc
    acc = seg

# Rust (fixed):
if acc.chars().count() < 5:  # character count, NOT bytes
    acc.push_str(&seg);
else {
    yield acc;
    acc = seg;
}
```

2. G2P (Grapheme-to-Phoneme)

Chinese text:

- Uses `pypinyin` for pinyin conversion
- Special handling for polyphones (多音字)
- Output: Pinyin with tone numbers (e.g., "ni3 hao3")

English text:

- Uses `g2p_en` for ARPAbet phonemes
- Handles unknown words via letter spelling
- Output: ARPAbet symbols (e.g., "HH AH0 L OW1")

Mixed Chinese/English:

- Uses `g2pw` model (ONNX) for context-aware disambiguation
- Critical for sentences like "这家restaurant的steak很有名"

3. BERT Encoding

Model: `chinese-roberta-wwm-ext-large`

- 24 layers, 1024 hidden size
- Input: Phone token IDs
- Output: Contextual embeddings [batch, seq_len, 1024]

Phone-to-BERT alignment:

- `word2ph` mapping: number of phones per word
- BERT outputs are duplicated to match phone count

4. T2S (GPT) Model

Architecture:

- Transformer decoder with cross-attention to BERT embeddings
- Vocabulary: ~1025 semantic tokens (codebook from SSL quantization)
- Special tokens: EOS (end of sequence)

Inference parameters: | Parameter | Default | Purpose | |-----------|---------|---------| | `top_k` | 15 | Keep top-k tokens | | `top_p` | 1.0 | Nucleus sampling threshold | | `temperature` | 1.0 | Sampling temperature | | `repetition_penalty` | 1.35 | Penalize repeated tokens |

5. VITS Decoder

Model: `SynthesizerTrn` (modified VITS architecture)

- Input: semantic codes + phone IDs + reference mel
- Output: 32kHz audio waveform
- Reference mel: 1025 bins extracted, sliced to 704 for VITS input

Key dimensions:

- Mel bins: 1025 (extracted) → 704 (VITS input, sliced from index 0)
- Sample rate: 32000 Hz

- Hop size: 640 samples (20ms frames)

Python vs Rust Implementation Mapping

File Mapping

Component	Python (primespeech)	Rust (gpt-sovits-mlx)
Main pipeline	`TTS_infer_pack/TTS.py`	`src/voice_clone.rs`
T2S model	`AR/models/t2s_model.py`	`src/voice_clone.rs` (inline)
Sampling utils	`AR/models/utils.py`	`src/voice_clone.rs`
Text processing	`text_segmentation_method.py`	`src/voice_clone.rs`
G2P	`g2pw/onnx_api.py`	`src/models/g2pw.rs`
VITS	`module/models.py`	`src/models/vits.rs` (MLX) / `src/models/vits_onnx.rs` (ONNX)

Step-by-Step Mapping

Step 1: Text Chunking

Python (`TTS.py` line 504-520):

```
def cut5(self, inp):
    # Split at every punctuation
    punds = r'[,.;?!、，。？！；：…"]'
    items = re.split(f'({punds})', inp)
    # Merge short segments
    mergeitems = []
    for i, seg in enumerate(items):
        if len(googletrans.convert(seg, 'zh')) < 5:
            # Merge with previous
```

Rust (`voice_clone.rs` line 1960-2010):

```rust
fn split_text_cut5(&self, text: &str) -> Vec<String> {
    let punct_pattern = Regex::new(r#"[,.;?!、，。？！；：…""''「」『』【】〔〕〈〉
《》\n]"#).unwrap();
    // Split and merge
    if acc.chars().count() < 5 {  // CRITICAL: chars(), not len()
        acc.push_str(&seg);
    }
}
```

Step 2: G2P Conversion

Python (`TTS.py` line 640-680):

```python
def get_phones_and_bert(self, text, language, ...):
    if language == "zh":
        phones, word2ph = g2p_chinese(text)
    elif language == "en":
        phones, word2ph = g2p_english(text)
    # Get BERT embeddings
    bert_emb = self.bert_model(phone_ids)
```

Rust (`voice_clone.rs` line 800-900):

```rust
fn get_phones_and_bert(&self, text: &str, language: &str) -> Result<(Vec<i32>,
Array)> {
    let (phones, word2ph) = match language {
        "zh" => self.g2p_chinese(text)?,
        "en" => self.g2p_english(text)?,
        _ => self.g2p_mixed(text)?,
    };
    let bert_emb = self.encode_bert(&phone_ids)?;
}
```

Step 3: Reference Audio Processing

Python (`TTS.py` line 350-400):

```python
def get_ref_spec(self, ref_audio_path):
    audio, sr = torchaudio.load(ref_audio_path)
    # Resample to 32kHz
    audio = torchaudio.functional.resample(audio, sr, 32000)
    # Extract mel spectrogram (1025 bins)
    spec = spectrogram_torch(audio, ...)  # [1, 1025, T]
    return spec
```

Rust (`voice_clone.rs` line 400-450):

```rust
fn extract_mel_spectrogram(&self, audio: &[f32]) -> Result<Array> {
    // STFT with 2048 FFT size, 640 hop
    let stft = self.compute_stft(audio)?;
    // Mel filterbank (1025 bins)
    let mel = self.apply_mel_filterbank(&stft)?;  // [1, 1025, T]
    Ok(mel)
}
```

Step 4: T2S Inference

Python (`t2s_model.py` line 575-750, `infer_panel_batch_infer`):

```python
def infer_panel_batch_infer(self, ...):
    # Encode prompt
    prompt = self.ar_text_embedding(prompt_ids) + self.ar_text_position(...)
    # Autoregressive loop
    for idx in range(max_len):
        logits = self.forward(...)
        # Sample next token
        samples = sample(logits, ...)
        if torch.argmax(logits, dim=-1)[0] == self.EOS or samples[0,0] ==
self.EOS:
            break
        y_emb = self.ar_audio_embedding(samples)
```

Rust (`voice_clone.rs` line 1300-1500):

```rust
fn t2s_inference(&self, phones: &[i32], bert_emb: &Array, prompt_codes: &[i32])
-> Result<Vec<i32>> {
    // Encode prompt
    let prompt_emb = self.ar_text_embedding(&prompt_ids)? +
self.ar_text_position(...)?;
    // Autoregressive loop
    loop {
        let logits = self.t2s_forward(...)?;
        let (token_id, argmax_token) = self.sample_token(&logits)?;
        // EOS check (CRITICAL: match Python exactly)
        if token_id == eos_token || argmax_token == eos_token {
            break;
        }
        generated.push(token_id);
    }
}
```

Step 5: Sampling

Python (`utils.py` line 110-162):

```python
def logits_to_probs(logits, top_k=None, top_p=None, temperature=1.0,
repetition_penalty=1.0):
    # Order: rep_penalty → top_p → temperature → top_k → softmax
    if repetition_penalty != 1.0:
        logits = apply_rep_penalty(logits)
    if top_p is not None and top_p < 1.0:
        logits = top_p_filter(logits, top_p)
    if temperature != 1.0:
        logits = logits / temperature
    if top_k is not None:
        logits = top_k_filter(logits, top_k)
    probs = F.softmax(logits, dim=-1)
    return probs

def sample(logits, ...):
    probs = logits_to_probs(logits, ...)
    idx = torch.multinomial(probs, num_samples=1)
    return idx
```

Rust (`voice_clone.rs` line 1662-1750):

```rust
fn sample_token(&self, logits: &Array) -> Result<(i32, i32)> {
    let mut logits_vec = logits.to_vec::<f32>()?;

    // 1. Repetition penalty
    self.apply_repetition_penalty(&mut logits_vec)?;

    // 2. Top-p filtering (on logits, before temperature)
    if self.top_p < 1.0 {
        self.apply_top_p_filter(&mut logits_vec)?;
    }

    // 3. Temperature scaling
    if self.temperature != 1.0 {
        for v in logits_vec.iter_mut() {
            *v /= self.temperature;
        }
    }

    // 4. Top-k filtering
    if self.top_k < logits_vec.len() {
        self.apply_top_k_filter(&mut logits_vec)?;
    }

    // 5. Softmax
    let probs = softmax(&logits_vec);

    // 6. Sample
    let token_id = categorical_sample(&probs)?;
    let argmax_token = argmax(&logits_vec);

    Ok((token_id, argmax_token))
}
```

Step 6: VITS Synthesis

Python (`TTS.py` line 904-920, speed==1.0 path):

```python
# Batched: concatenate all chunks
all_codes = torch.cat([r['pred_semantic'] for r in chunk_results], dim=-1)
all_phones = torch.cat([r['phones'] for r in chunk_results], dim=-1)

# Single VITS call
audio = self.vits_model.decode(
    codes=all_codes,      # [1, 1, T_total_codes]
    text=all_phones,      # [1, T_total_phones]
    refer=ref_mel[:, :704, :],  # [1, 704, T_mel] - sliced!
)
```

Rust with ONNX (`voice_clone.rs` line 700-800):

```rust
if let Some(ref vits_onnx) = self.vits_onnx {
    // Batched: concatenate all chunks (matching Python)
    let all_tokens: Vec<i32> = chunk_results.iter()
        .flat_map(|r| r.semantic_tokens.iter().copied())
        .collect();
    let all_phones: Vec<i32> = chunk_results.iter()
        .flat_map(|r| r.phone_ids.iter().copied())
        .collect();

    // Single ONNX VITS call
    let audio = vits_onnx.decode(
        &all_tokens,
        &all_phones,
        &ref_mel_704,  // Sliced to 704 bins
    )?;
}
```

Step 7: Audio Postprocessing

Python (`TTS.py` line 991-1027):

```python
def audio_postprocess(self, audio_list, sr):
    output = []
    for audio in audio_list:
        # Clip to [-1, 1]
        audio = torch.clamp(audio, -1.0, 1.0)
        # Append 0.3s silence
        silence = torch.zeros(int(sr * 0.3))
        audio = torch.cat([audio, silence])
        output.append(audio)
    # Concatenate all
    return torch.cat(output)
```

Rust (`voice_clone.rs` line 750-800):

```rust
fn postprocess_audio(&self, raw_samples: &[f32], chunk_results: &[ChunkResult])
-> Vec<f32> {
    let silence_samples = (0.3 * 32000.0) as usize;  // 0.3s at 32kHz
    let mut all_samples = Vec::new();

    for result in chunk_results {
        let chunk_audio = &raw_samples[result.audio_start..result.audio_end];

        // Clip to [-1, 1]
        let clipped: Vec<f32> = chunk_audio.iter()
            .map(|s| s.clamp(-1.0, 1.0))
            .collect();

        all_samples.extend_from_slice(&clipped);
        // Append 0.3s silence
        all_samples.extend(std::iter::repeat(0.0f32).take(silence_samples));
    }

    all_samples
}
```

Critical Fixes for Mixed Chinese/English

Fix 1: EOS Detection Logic

Problem: Rust T2S generated ~2x more tokens than Python for the same input, causing noise insertions.

Root Cause: Overly strict EOS detection in Rust.

Python (correct, `t2s_model.py` line 871):

```python
# Simple: EITHER condition triggers EOS
if torch.argmax(logits, dim=-1)[0] == self.EOS or samples[0, 0] == self.EOS:
    stop = True
```

Rust (before fix):

```rust
// WRONG: Required BOTH conditions when below target
let eos_detected = if generated_count < target_tokens {
    token_id == eos_token && argmax_token == eos_token  // Too strict!
} else {
    token_id == eos_token || argmax_token == eos_token
};

// Also had min_tokens threshold that blocked early EOS
if generated_count < min_tokens && eos_detected {
    // Re-sample with EOS masked out (WRONG!)
    continue;
}
```

Rust (after fix, `voice_clone.rs` line 1438-1457):

```rust
// Match Python exactly: EITHER condition triggers EOS
let eos_detected = token_id == eos_token || argmax_token == eos_token;
if eos_detected {
    break;  // No min_tokens check, no re-sampling
}
```

Impact:

- Before: 376 tokens for English sentence → noise insertions
- After: 212 tokens (matches Python's 207) → clean audio

Fix 2: Sampling Order

Problem: Subtle quality differences in generated tokens.

Root Cause: Rust applied softmax before top-k filtering.

Correct order (Python `utils.py`):

```
1. Repetition penalty (on logits)
2. Top-p filtering (on logits)
3. Temperature scaling (on logits)
4. Top-k filtering (on logits)
5. Softmax (logits → probabilities)
6. Categorical sample
```

Rust (before fix):

```rust
// WRONG: softmax was applied before top-k
let probs = softmax(&logits);  // Too early!
let top_k_probs = apply_top_k(&probs);  // Wrong: should filter logits
```

Rust (after fix):

```rust
// Correct order matching Python
apply_repetition_penalty(&mut logits);
apply_top_p_filter(&mut logits);      // On logits
apply_temperature(&mut logits);        // On logits
apply_top_k_filter(&mut logits);       // On logits (set others to -inf)
let probs = softmax(&logits);          // Now convert to probs
let token = categorical_sample(&probs);
```

Fix 3: Text Chunking Merge Threshold

Problem: Mixed Chinese/English text was split differently than Python.

Root Cause: Rust used byte length instead of character count.

Python (`cut5`):

```python
if len(segment) < 5:  # len() counts characters for str
    merge_with_previous()
```

Rust (before fix):

```rust
if acc.len() < 5 {  // WRONG: len() counts bytes, not chars
    // "你好" is 6 bytes but 2 characters
}
```

Rust (after fix):

```rust
if acc.chars().count() < 5 {  // Correct: count Unicode characters
    acc.push_str(&seg);
}
```

Impact:

- Chinese characters are 3 bytes each in UTF-8
- "你好" would be 6 bytes (>5) but only 2 characters (<5)
- Incorrect merging led to different chunk boundaries

Fix 4: VITS Batched Decode

Problem: Per-chunk VITS decode caused noise at chunk boundaries.

Root Cause: Each chunk decoded in isolation loses context from adjacent chunks.

Python (speed==1.0):

```python
# Concatenate ALL chunks before VITS
all_codes = concat([chunk.codes for chunk in chunks])
all_phones = concat([chunk.phones for chunk in chunks])
audio = vits.decode(all_codes, all_phones, ref_mel)  # Single call
```

Rust (before fix, MLX VITS):

```rust
// Per-chunk decode
for chunk in chunks {
    let audio = vits.decode(chunk.codes, chunk.phones, ref_mel)?;
    outputs.push(audio);
}
// Concatenate audio (but chunk boundaries have artifacts)
```

Rust (after fix, ONNX VITS):

```rust
// Match Python: batched decode
let all_codes = chunks.iter().flat_map(|c| c.codes.iter()).collect();
let all_phones = chunks.iter().flat_map(|c| c.phones.iter()).collect();
let audio = vits_onnx.decode(&all_codes, &all_phones, &ref_mel)?;
// Split by chunk boundaries, add silence
```

Fix 5: Audio Postprocessing

Problem: Per-chunk tail trimming created audio artifacts.

Root Cause: Trimming based on energy detection was removing valid audio.

Python (no trimming):

```python
def audio_postprocess(audio_list):
    for audio in audio_list:
        audio = torch.clamp(audio, -1.0, 1.0)  # Just clip
        audio = torch.cat([audio, silence_0.3s])  # Add silence
    return concat(audio_list)
```

Rust (before fix):

```rust
// WRONG: Energy-based tail trimming
let trim_point = find_last_significant_sample(audio, threshold=0.01);
audio.truncate(trim_point);
```

Rust (after fix):

```rust
// Match Python: no trimming, just clip + silence
for sample in audio.iter_mut() {
    *sample = sample.clamp(-1.0, 1.0);
}
audio.extend(silence_0.3s);
```

Audio Quality Issues and Solutions

Issue 1: Noise Insertions in English Phrases

Symptom: Random noise/artifacts within long English phrases like "The Economist, Weekly Commercial Times, Bankers' Gazette, and Railway Monitor."

Cause: Over-generation of semantic tokens due to strict EOS detection.

Solution: Fix EOS detection to match Python (see Fix 1).

Issue 2: Noise at Punctuation Boundaries

Symptom: Clicking or popping sounds at commas and periods within chunks.

Cause: Per-chunk VITS decode loses cross-chunk context.

Solution: Use batched VITS decode (see Fix 4).

Issue 3: Trailing Noise

Symptom: Noise at the end of generated audio.

Cause: Energy-based tail trimming was too aggressive or inconsistent.

Solution: Remove trimming, use simple clip + silence (see Fix 5).

Issue 4: Different Chunk Boundaries for Mixed Text

Symptom: Audio pacing differs between Python and Rust for the same text.

Cause: Byte vs character count for merge threshold.

Solution: Use character count (see Fix 3).

Verification Checklist

When implementing GPT-SoVITS in a new language/framework:

- [ ] Text chunking uses character count, not byte length
- [ ] G2P handles mixed language correctly (use g2pw for Chinese+English)
- [ ] BERT embeddings are duplicated per `word2ph` mapping
- [ ] T2S EOS detection: `argmax == EOS OR sampled == EOS` (no AND)
- [ ] Sampling order: rep_penalty → top_p → temperature → top_k → softmax
- [ ] VITS input mel is sliced to 704 bins (not full 1025)
- [ ] Batched VITS decode for speed==1.0 path
- [ ] Audio postprocess: clip to [-1,1], append 0.3s silence per chunk, no trimming

---

Appendix: Token Count Comparison

For the English sentence:

> "The Economist, Weekly Commercial Times, Bankers' Gazette, and Railway Monitor. A Political, Literary and General Newspaper."

Chunk	Python Tokens	Rust (before)	Rust (after)
"The Economist,"	25	73	26
"Weekly Commercial Times,"	30	72	31
"Bankers' Gazette,"	36	41	35
"and Railway Monitor."	28	45	29
"A Political,"	36	38	35
"Literary and General Newspaper."	52	107	56
Total	**207**	**376**	**212**
Duration	9.52s	17.3s	9.80s

The fix reduced Rust's over-generation from 1.8x to 1.02x of Python.