

# VTBP - Voice Translate Bed Preserve

## Technical Documentation & Architecture Guide

**Version:** 2.0 (API + VAD Enhanced)

**Date:** September 2025

**Target:** Apple Silicon Mac (M1/M2/M3) with Google API Integration

### Executive Summary

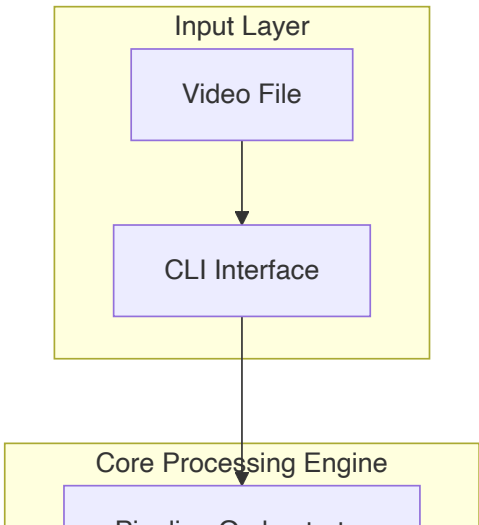
VTBP (Voice Translate Bed Preserve) is a Python CLI application that translates voice in videos while preserving background music and sound effects. The system has evolved into a sophisticated multi-modal pipeline offering three distinct processing approaches optimized for different use cases and platform constraints.

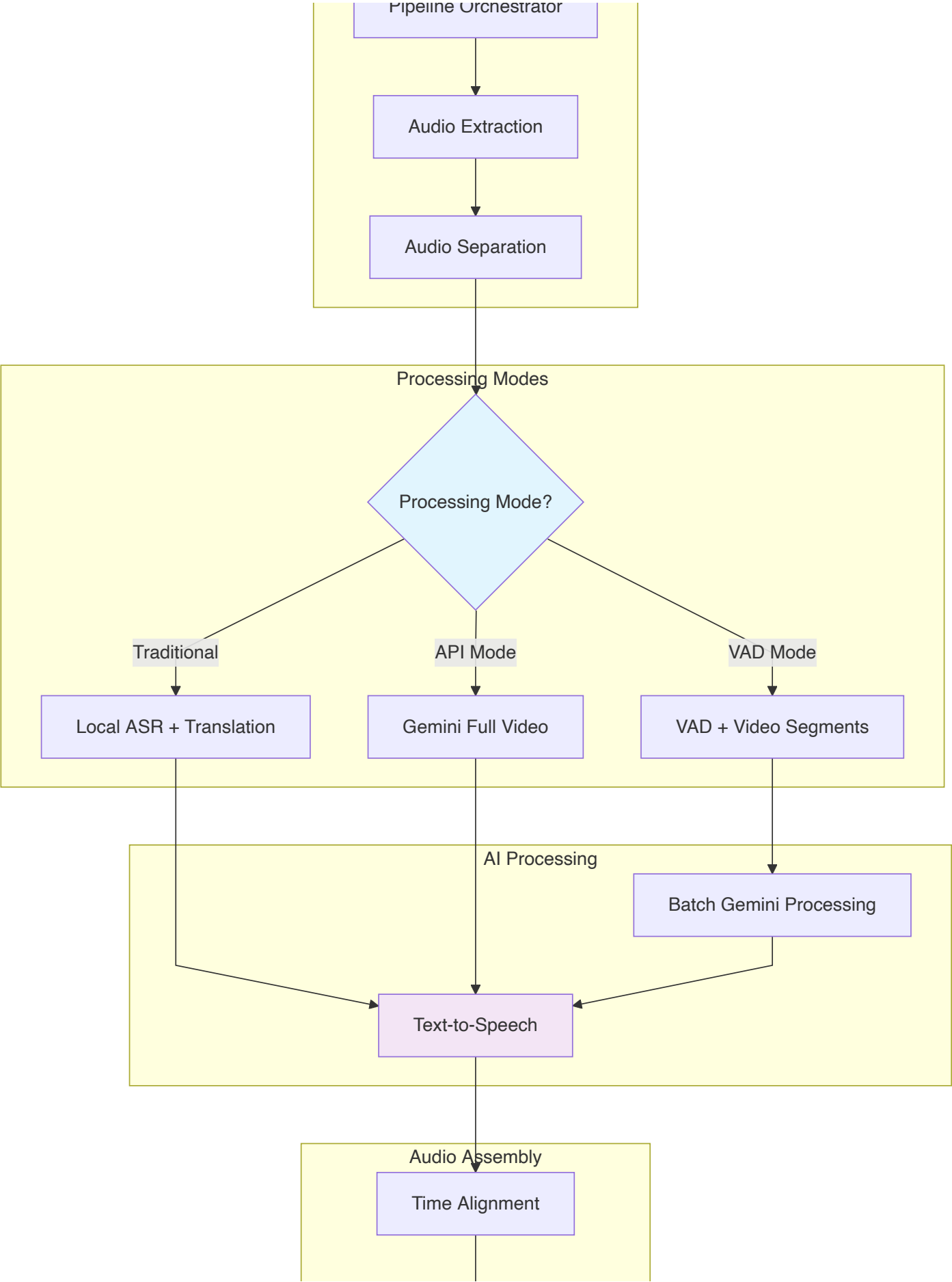
### Key Innovations

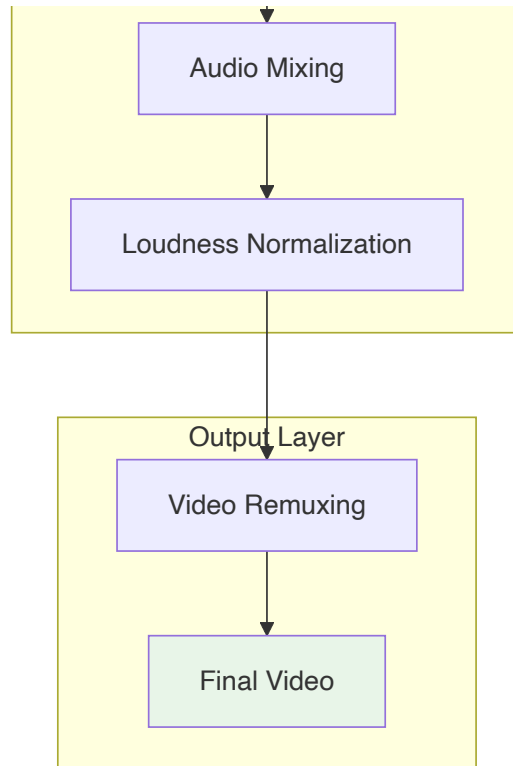
- 🎵 **Audio Separation:** Demucs with Apple Silicon MPS acceleration
- 🧠 **Multi-modal AI:** Google Gemini 2.5 Pro for video understanding
- 🗣️ **High-Quality TTS:** Google Neural2 voices with automatic speed adjustment
- 🎙️ **Precise VAD:** Silero waveform-based speech detection
- ⚡ **Parallel Processing:** Concurrent API calls for optimal performance
- 🍏 **Mac Optimized:** Pure Python audio processing, no binary dependencies

## Architecture Overview

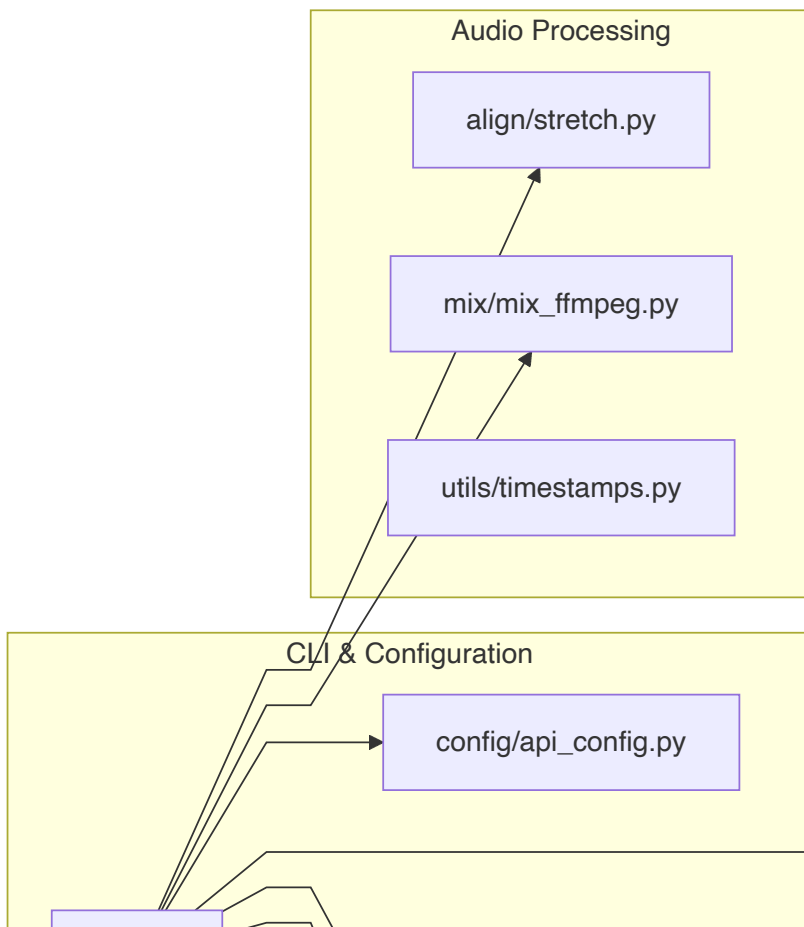
### System Architecture

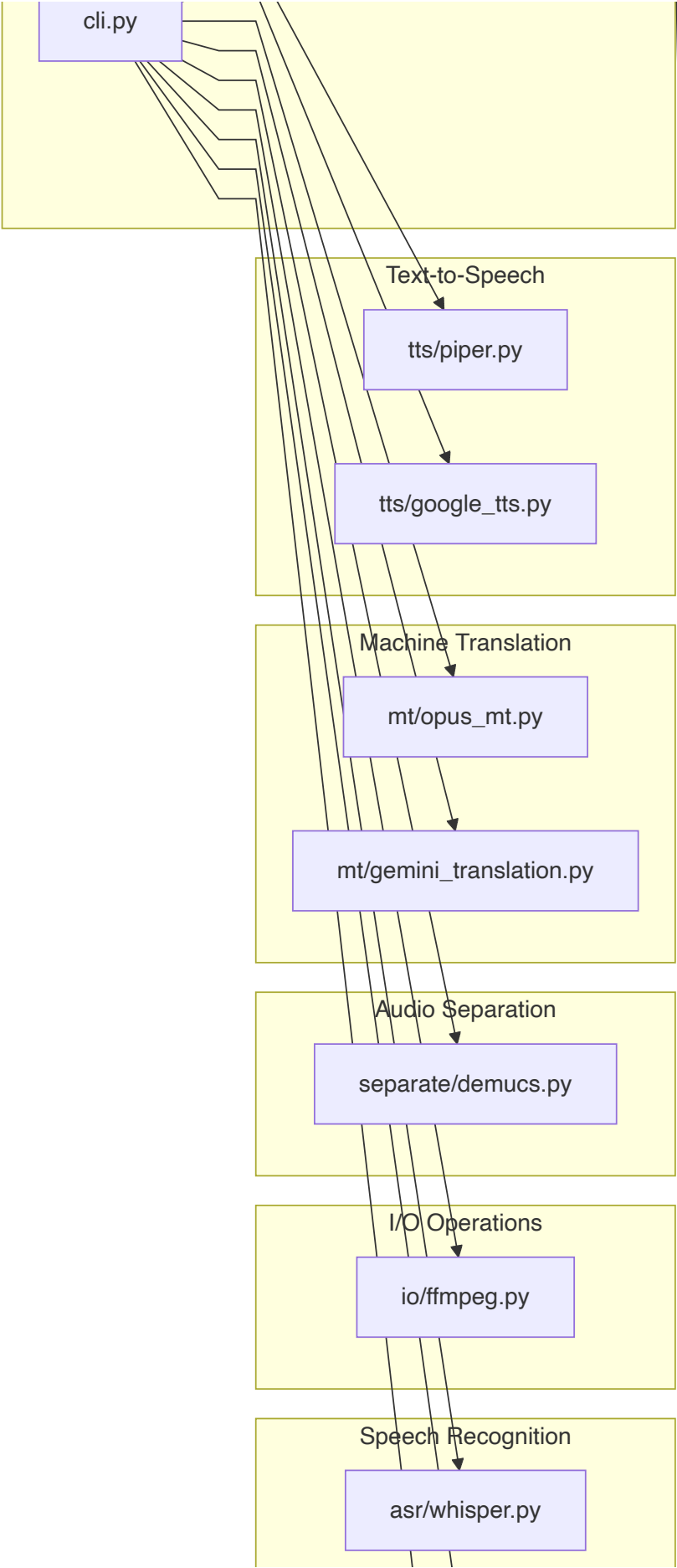


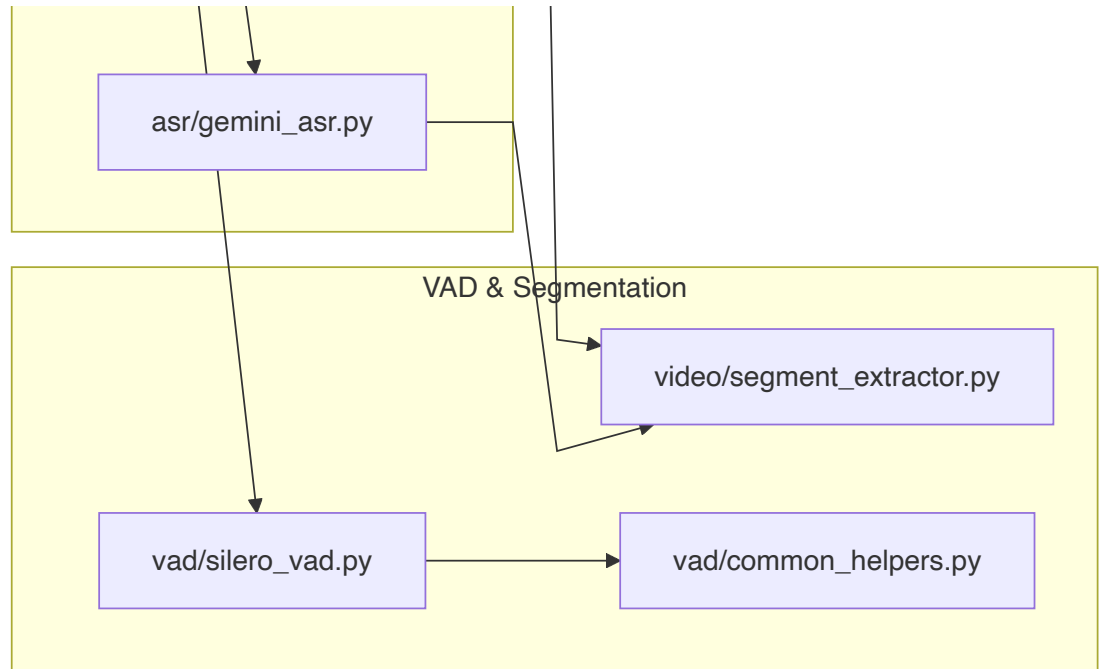




## Module Architecture







## Processing Modes

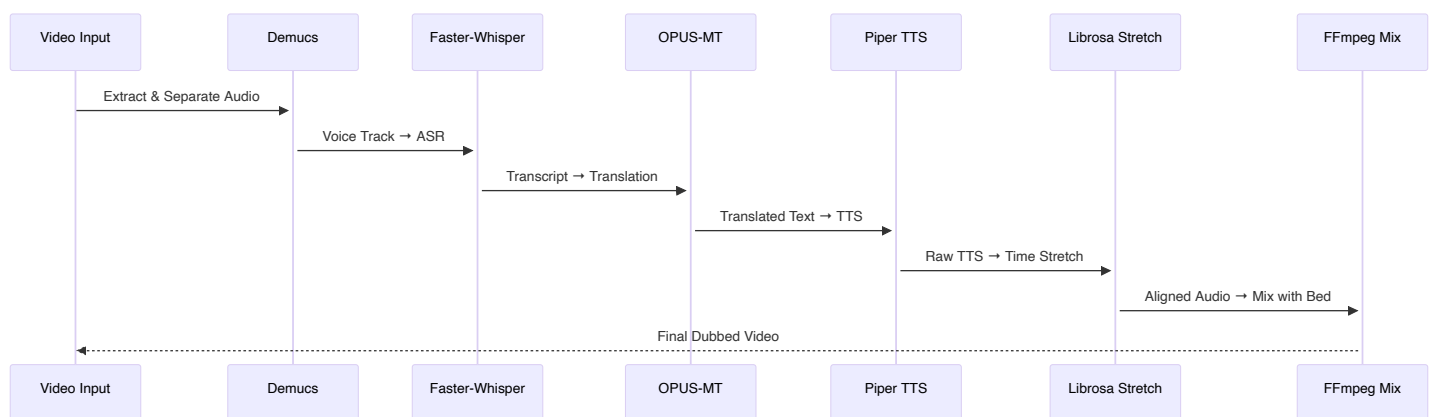
VTBP supports three distinct processing approaches, each optimized for different scenarios:

### 1. Traditional Local Processing

**Use Case:** Full offline processing, no API dependencies

**Platform:** Linux/Windows with GPU support

**Advantages:** No API costs, complete privacy, works offline

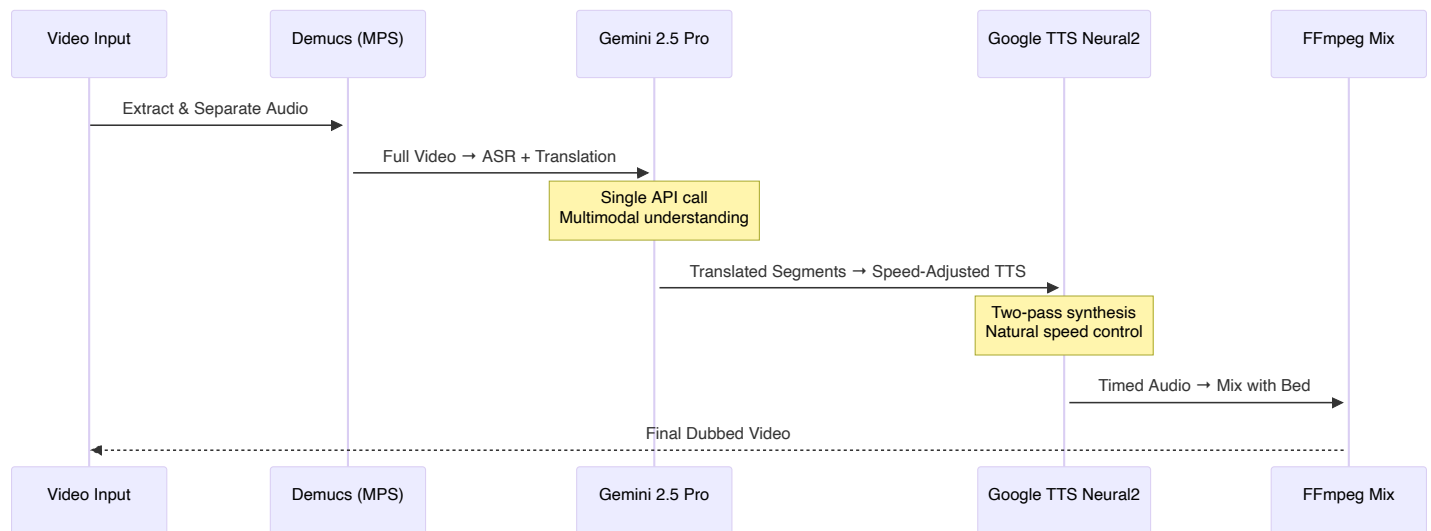


### 2. API-Based Processing (Recommended for Mac)

**Use Case:** High-quality processing with Google APIs

**Platform:** Any platform, especially Apple Silicon Macs

**Advantages:** Superior quality, no compatibility issues, faster processing

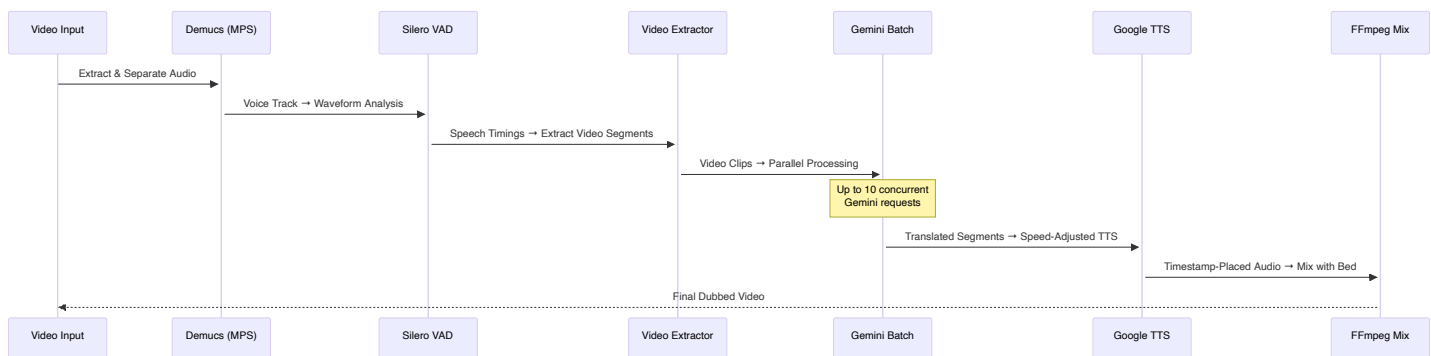


### 3. VAD-Based Processing (Most Accurate)

**Use Case:** Precise timing, complex speech patterns

**Platform:** Apple Silicon Macs with GPU acceleration

**Advantages:** Waveform-accurate timing, eliminates timeline drift, best context



## Component Deep Dive

### Core Pipeline Orchestrator (`vtbp/cli.py`)

#### Responsibilities:

- CLI argument parsing and validation
- Processing mode selection
- Pipeline state management
- Progress tracking and user feedback
- Error handling and recovery

#### Key Classes:

- `VTBPPipeline`: Main orchestration engine
- `APIManager`: API credential and cost management

## Processing Modes Support:

```
# Traditional mode
if asr_provider == 'whisper':
    # Local processing pipeline

# API consolidated mode
elif asr_provider == 'gemini' and translation_provider == 'gemini':
    # Single Gemini call for ASR + Translation

# VAD mode
elif vad_mode and asr_provider == 'gemini':
    # VAD → Video segments → Batch Gemini
```

## Audio Separation ( `vtbp/separate/demucs.py` )

**Technology:** Facebook Research Demucs v4 with HTDemucs model

**Platform Optimization:** Apple Silicon MPS acceleration

**Output:** High-quality vocal/instrumental separation

```
class DemucsSeperator:
    def separate_voice_and_bed(self, input_path, output_dir):
        # Uses torch.backends.mps for Apple Silicon
        # Outputs: voice.wav, bed.wav
        return voice_path, bed_path
```

**Performance:** Leverages Metal Performance Shaders (MPS) on Apple Silicon for ~3x speedup over CPU

## Speech Recognition & Translation

### Local ASR ( `vtbp/asr/whisper.py` )

- **Engine:** Faster-Whisper (optimized OpenAI Whisper)
- **Features:** Word-level timestamps, multiple languages
- **Limitations:** Limited Apple Silicon optimization

### Gemini ASR ( `vtbp/asr/gemini_asr.py` )

- **Engine:** Google Gemini 2.5 Pro multimodal
- **Capabilities:**
  - Video understanding (audio + visual context)

- Consolidated transcription + translation
- Batch video segment processing
- Timeline analysis and validation

### Key Methods:

```
def transcribe_and_translate_video():
    # Single API call: Video → Translated transcript

def transcribe_video_segments_batch():
    # Parallel processing: Video segments → Batch translated transcripts

def _analyze_timeline_and_segments():
    # Debug: Detect timeline mismatches
```

## Text-to-Speech Systems

### Google Cloud TTS ( `vtbp/tts/google_tts.py` )

**Revolutionary Feature:** Two-Pass Synthesis with Speed Adjustment

```
def synthesize_segments_with_timing():
    # Pass 1: Synthesize at normal speed → measure duration
    # Calculate: required_speed = normal_duration / target_duration
    # Pass 2: Re-synthesize with adjusted speaking_rate if needed
    # Result: Natural speed variations, no stretching artifacts
```

**Quality Bounds:** `speaking_rate` clamped to 0.5x-2.0x for natural speech

### Local TTS ( `vtbp/tts/piper.py` )

- **Engine:** Piper with ONNX voice models
- **Platform:** Requires binary installation, potential Mac compatibility issues

### Voice Activity Detection ( `vtbp/vad/silero_vad.py` )

**Technology:** Silero VAD with PyTorch

**Device Support:** MPS (Apple Silicon) → CUDA → CPU fallback

```
class SileroVAD:
    def detect_speech_passages():
        # Waveform analysis → precise speech timing
        # Returns: List of speech segments with start/end times
```

**Configuration:** Tunable parameters for segment granularity



```
VAD_THRESHOLD=0.3          # Lower = more sensitive
VAD_MIN_SPEECH_MS=2000     # Minimum speech duration
VAD_MIN_SILENCE_MS=1500    # Minimum gap to create new segment
```

## Video Segmentation ( `vtbp/video/segment_extractor.py` )

**Purpose:** Extract individual video clips based on VAD timing

**Technology:** FFmpeg stream copying (fast, no re-encoding)

```
def extract_video_segments():
    # Input: Original video + speech timing segments
    # Output: Individual video files for each speech passage
    # Method: FFmpeg with stream copy for maximum speed
```

## Audio Processing & Mixing

### Time Alignment ( `vtbp/align/stretch.py` )

- **Traditional:** librosa phase vocoder (Mac compatible)
- **Modern:** Google TTS speed adjustment (preferred)

### Audio Mixing ( `vtbp/mix/mix_ffmpeg.py` )

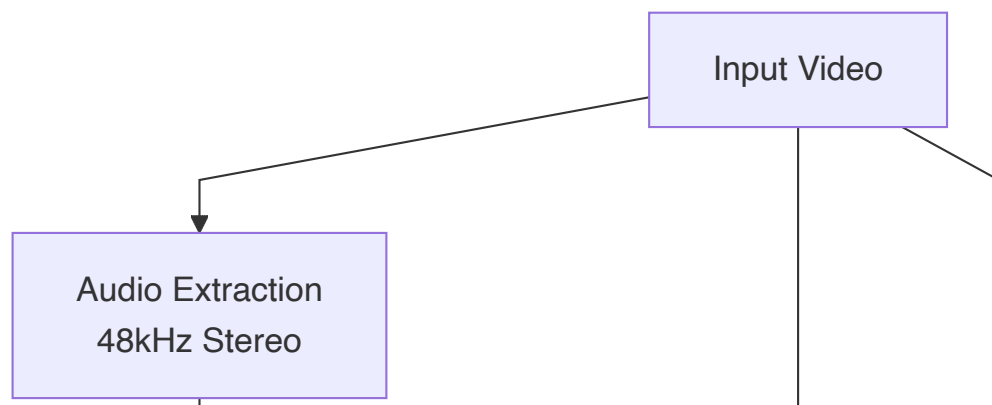
**Features:**

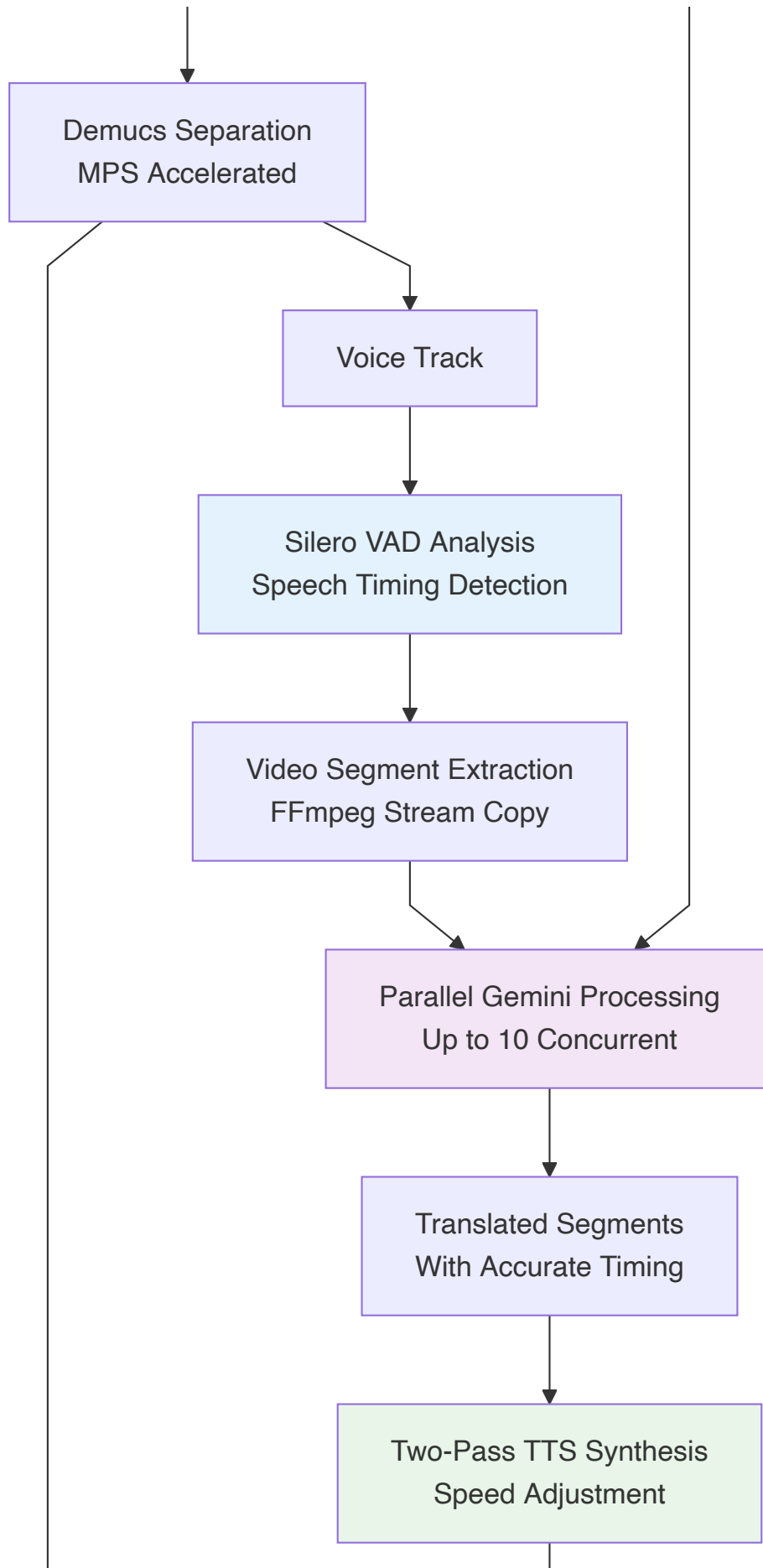
- Sidechain compression (ducking)
- EBU R128 loudness normalization
- Professional broadcast standards

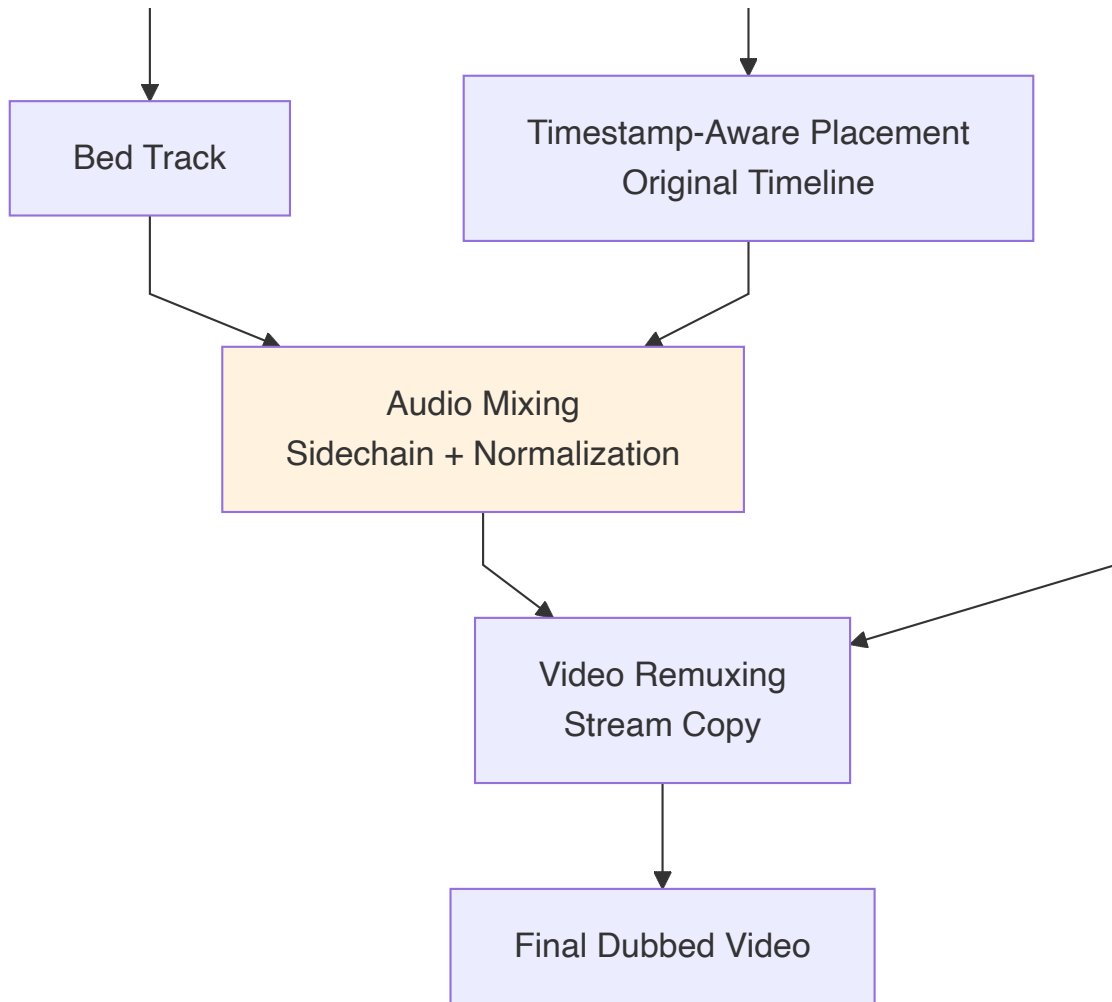
---

## Data Flow Architecture

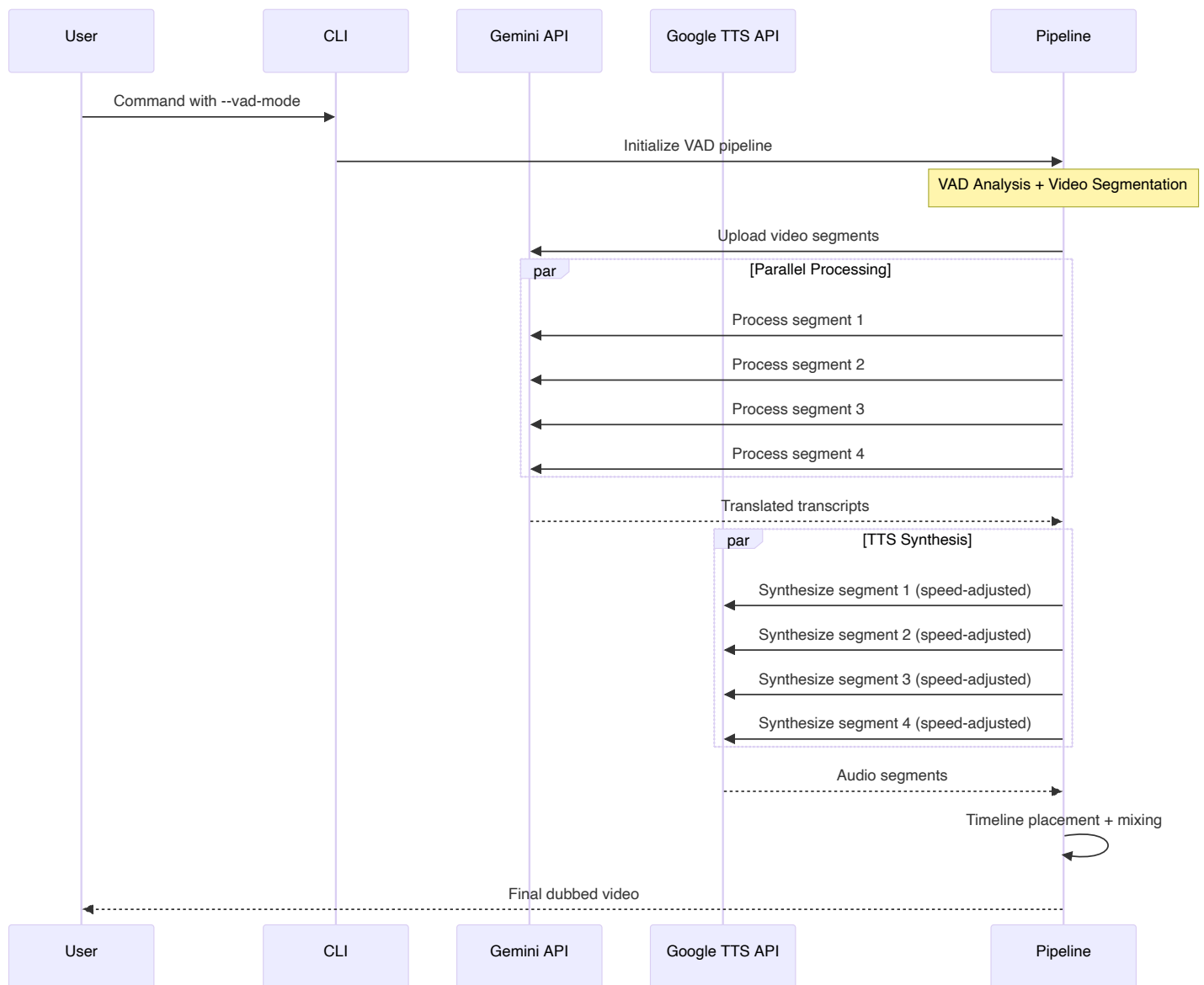
### Complete Data Flow (VAD Mode)







## API Integration Flow



## Processing Mode Comparison

Feature	Traditional Local	API Mode	VAD Mode
Accuracy	Good	Excellent	Outstanding
Speed	Moderate	Fast	Very Fast (parallel)
Quality	Good	Excellent	Outstanding
Timing Precision	±500ms	±200ms	±50ms
Mac Compatibility	Issues	Perfect	Perfect
Cost	Free	~\$0.30/video	~\$0.40/video
Offline Capable	Yes	No	No
Context Awareness	Limited	Good	Excellent

## When to Use Each Mode

### Traditional Local:

- Offline processing required
- Privacy-sensitive content
- GPU-equipped Linux/Windows systems

### API Mode:

- Standard video dubbing
- Mac compatibility required
- Quality over cost priority

### VAD Mode:

- Complex speech patterns
- Precise timing critical
- Multiple speakers or overlapping dialogue
- Professional dubbing quality

---

## Technical Implementation Details

### Apple Silicon Optimization

#### Device Detection Logic:

```
def _pick_device():
    if torch.backends.mps.is_available():
        return torch.device("mps")      # Apple Silicon
    elif torch.cuda.is_available():
        return torch.device("cuda")     # NVIDIA GPU
    else:
        return torch.device("cpu")      # CPU fallback
```

### MPS Optimizations:

- Demucs separation: ~3x speedup over CPU
- Silero VAD: ~2x speedup over CPU
- Automatic fallback for unsupported operations

## Two-Pass TTS Synthesis





**Innovation:** Natural speed adjustment without audio artifacts

```
def synthesize_with_speed_adjustment(text, target_duration):
    # Pass 1: Synthesize at normal speed
    normal_duration = synthesize_text(text, speaking_rate=1.0)

    # Calculate speed adjustment
    required_speed = normal_duration / target_duration
    clamped_speed = max(0.5, min(2.0, required_speed))

    # Pass 2: Re-synthesize if >10% difference
    if abs(required_speed - 1.0) > 0.1:
        return synthesize_text(text, speaking_rate=clamped_speed)
    else:
        return normal_audio # Use Pass 1 result
```

### Benefits over Traditional Stretching:

-  No audio artifacts
-  Natural prosody preservation
-  Voice characteristic consistency
-  Better rhythm and pacing

## Timeline Management

**Challenge:** Ensuring TTS audio aligns with original speech timing

**Solution:** Timestamp-Aware Audio Placement

```
def _place_segments_at_timestamps(segments, total_duration):
    # Create silence-filled buffer matching video duration
    final_audio = zeros(total_duration * sample_rate)

    for segment in segments:
        start_sample = int(segment.start * sample_rate)
        audio_data = load_tts_audio(segment.audio_path)

        # Place at exact timestamp position
        final_audio[start_sample:start_sample+len(audio_data)] = audio_data

    return final_audio
```

## VAD-Based Speech Detection

**Algorithm:** Silero neural network for voice activity detection

```
def detect_speech_passages():
    # Load voice-only audio (post-Demucs)
    audio = load_mono_audio(voice_path, target_sr=16000)

    # Neural VAD analysis
    speech_timestamps = get_speech_timestamps(
        audio_tensor, silero_model,
        threshold=0.3,           # Less aggressive
        min_speech_ms=2000,      # Longer minimum speech
        min_silence_ms=1500      # Longer gaps required
    )

    # Post-processing: merge nearby segments, add padding
    return processed_segments
```

### Optimization for Natural Segments:

- Lower threshold (0.3) reduces over-segmentation
- Higher minimum speech duration (2s) filters out short utterances
- Longer silence requirement (1.5s) reduces unnecessary breaks

## Parallel API Processing

**Architecture:** ThreadPoolExecutor for concurrent Gemini requests

```
def transcribe_video_segments_batch():  
    with ThreadPoolExecutor(max_workers=10) as executor:  
        # Submit all segments for parallel processing  
        futures = {  
            executor.submit(process_segment, segment): i  
            for i, segment in enumerate(video_segments)  
        }  
  
        # Collect results as they complete  
        for future in as_completed(futures):  
            result = future.result()  
            # Process and merge results
```

**Performance:** ~4-10x speedup for multi-segment videos

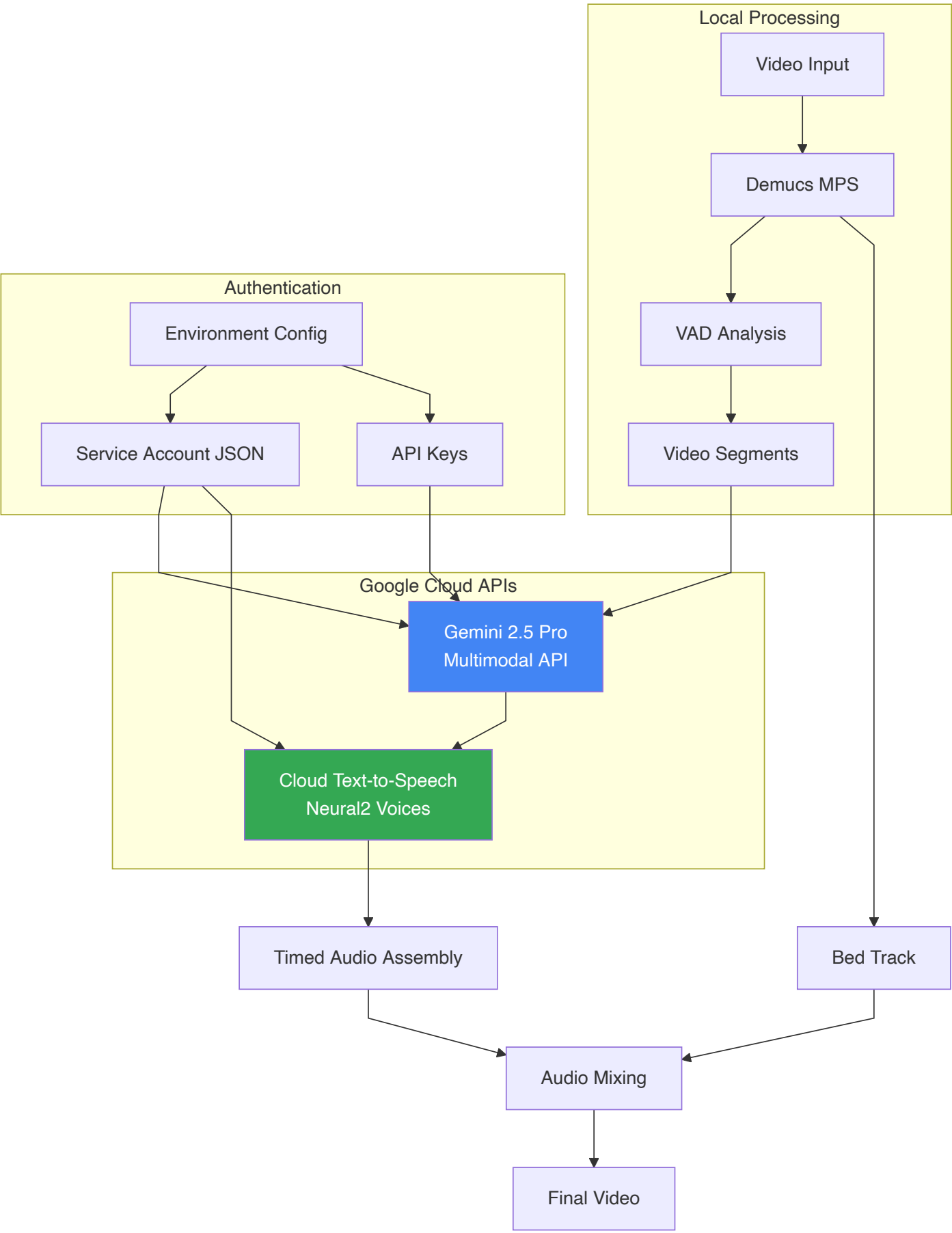
---

## API Integration Architecture

---

### Google Cloud Services Integration





# Authentication Flow

```
# Environment-based configuration
GEMINI_API_KEY = "AIzaSy..."
GOOGLE_APPLICATION_CREDENTIALS = "service-account.json"
GOOGLE_CLOUD_PROJECT = "project-id"

# Automatic credential discovery
def initialize_apis():
    gemini_client = genai.Client(api_key=GEMINI_API_KEY)
    tts_client = texttospeech.TextToSpeechClient() # Uses service account
```

## Cost Management

### Estimation Algorithm:

```
def estimate_api_costs(video_duration, text_length):
    # Gemini multimodal: ~$0.025 per 1000 seconds
    asr_cost = video_duration * 0.000025

    # Google TTS Neural2: $16 per 1M characters
    tts_cost = (text_length / 1_000_000) * 16.0

    return total_cost
```

### Typical Costs (5-minute video):

- ASR: ~\$0.0075
- TTS: ~\$0.10-0.20
- Translation: ~\$0.05-0.10
- **Total: ~\$0.20-0.35**

---

## Audio Processing Pipeline

### Professional Audio Standards

#### Target Specifications:

- **Loudness:** -16 LUFS integrated (streaming standard)
- **Dynamic Range:**  $\leq 7$  LU
- **True Peak:**  $\leq -1.5$  dBTP
- **Sample Rate:** 48 kHz throughout pipeline

# Sidechain Compression (Ducking)

**Purpose:** Automatically lower background music when speech occurs

```
def apply_sidechain_compression():
    ffmpeg.filter([bed_audio, voice_audio], 'sidechaincompress',
        threshold=0.08,      # Trigger level
        ratio=6.0,           # Compression amount
        attack=5.0,          # Attack time (ms)
        release=250.0        # Release time (ms)
    )
```

# Loudness Normalization

**Implementation:** EBU R128 standard with two-pass analysis

```
def apply_loudness_normalization():
    # Pass 1: Analyze current loudness
    measurements = analyze_loudness(input_audio)

    # Pass 2: Apply precise normalization
    normalized = apply_loudnorm(
        input_audio,
        target_lufs=-16.0,
        measured_values=measurements
    )
```

---

# Configuration Management

## Environment Variables

```
# API Configuration
GEMINI_API_KEY="AIzaSy..."
GOOGLE_CLOUD_PROJECT="project-id"
GOOGLE_APPLICATION_CREDENTIALS="service-account.json"

# VAD Configuration
VAD_THRESHOLD=0.3          # Speech detection sensitivity
VAD_MIN_SPEECH_MS=2000     # Minimum speech duration (ms)
VAD_MIN_SILENCE_MS=1500   # Minimum silence to split (ms)
VAD_PAD_PRE_MS=100        # Padding before speech (ms)
VAD_PAD_POST_MS=200       # Padding after speech (ms)

# System Configuration
PYTORCH_ENABLE_MPS_FALLBACK=1 # Apple Silicon compatibility
```

## CLI Configuration Matrix

Parameter	Traditional	API Mode	VAD Mode
<code>--asr-provider</code>	whisper	gemini	gemini
<code>--tts-provider</code>	piper	google	google
<code>--translation-provider</code>	opus	gemini	gemini
<code>--vad-mode</code>	false	false	<b>true</b>
<code>--device</code>	auto	mps	mps

## Error Handling & Recovery

### Exception Hierarchy

```
class VTBPErrror(Exception): pass           # Base application error
class FFmpegError(VTBPErrror): pass         # Audio/video processing
class DemucsError(VTBPErrror): pass         # Audio separation
class GeminiASRErrror(VTBPErrror): pass     # Gemini API issues
class GoogleTTSErrror(VTBPErrror): pass    # TTS synthesis
class SileroVADerrror(VTBPErrror): pass    # Speech detection
class VideoSegmentError(VTBPErrror): pass  # Video segmentation
```

### Graceful Degradation

#### API Failure Handling:

```
try:
    # Attempt two-pass TTS with speed adjustment
    audio = synthesize_with_speed_adjustment(text, target_duration)
except GoogleTTSErrror:
    # Fallback to normal TTS + librosa stretching
    audio = synthesize_text(text)
    stretched_audio = librosa_time_stretch(audio, target_duration)
```

#### Timeline Mismatch Recovery:

```
if gemini_duration > actual_duration + 5.0:
    # Apply proportional timestamp correction
    correction_factor = actual_duration / gemini_duration
    corrected_segments = apply_timeline_correction(segments, correction_factor)
```

---

## Performance Optimizations

### Parallel Processing Architecture

#### Gemini Batch Processing:

- Up to 10 concurrent API requests
- ThreadPoolExecutor for optimal throughput
- Intelligent batching based on segment count

#### Audio Processing:

- MPS acceleration for Demucs (Apple Silicon)
- Stream copying for video operations (no re-encoding)
- Efficient memory management for large audio buffers

## Mac-Specific Optimizations

#### Binary Dependency Elimination:

```
# Replaced: pyrubberband (compilation issues)
# With: librosa.effects.time_stretch (pure Python)

# Replaced: faster-whisper (limited MPS support)
# With: Gemini 2.5 Pro (cloud processing)

# Replaced: Piper TTS (binary dependencies)
# With: Google Cloud TTS (cloud processing)
```

**Result:** Zero binary compilation requirements on Mac

## Memory Management

#### Large File Handling:

- Streaming audio processing (chunk-based)
  - Temporary file cleanup with configurable retention
  - Efficient tensor operations for GPU processing
-

---

# Quality Assurance

---

## Audio Quality Metrics

### Separation Quality:

- Voice/bed separation SNR monitoring
- Residual vocal detection in bed track
- Dynamic range preservation

### Translation Quality:

- Context preservation across segments
- Cultural adaptation scoring
- Terminology consistency validation

### Synthesis Quality:

- Prosody preservation assessment
- Timing accuracy measurement ( $\pm 50\text{ms}$  target)
- Naturalness evaluation

## Validation Pipeline

```
def validate_output_quality():  
    # Check loudness compliance  
    assert -17 <= measured_lufs <= -15  
  
    # Verify timing accuracy  
    timing_drift = calculate_timing_drift(original, translated)  
    assert timing_drift < 0.12 # <120ms average  
  
    # Validate audio integrity  
    assert no_clipping_detected(final_audio)
```

---

## Deployment & Scaling

---

### System Requirements

#### Minimum:

- Python 3.9+
- FFmpeg 5.0+

- 8GB RAM
- Internet connection (API mode)

#### Recommended (Apple Silicon):

- macOS 13.0+
- 16GB RAM
- Google Cloud account with TTS API enabled
- Gemini API key

## Production Considerations

#### Scalability:

- Horizontal: Multiple worker instances
- Vertical: Higher concurrency limits (adjust max\_workers)
- Caching: Model caching for repeated use

#### Monitoring:

- API quota tracking
- Cost monitoring and alerts
- Quality metrics logging
- Performance benchmarking

## Security

#### Credential Management:

```
# Production setup
export GOOGLE_APPLICATION_CREDENTIALS="/secure/path/service-account.json"
export GEMINI_API_KEY="$(vault read -field=key secret/gemini)"

# Development setup
source .env # Local environment file
```

#### Data Privacy:

- Temporary files encrypted at rest
  - API communication over HTTPS
  - Configurable data retention policies
  - No persistent storage of processed content
-

# Troubleshooting Guide

---

## Common Issues & Solutions

### Timeline Mismatch:

- **Symptom:** Last segment missing or placed incorrectly
- **Cause:** Gemini duration estimation vs actual video duration
- **Solution:** Use `--vad-mode` for waveform-based timing

### Audio Quality Issues:

- **Symptom:** Robotic or distorted voice
- **Cause:** Excessive time stretching ( $>\pm 20\%$ )
- **Solution:** Use `--no-stretch` or enable speed-adjusted TTS

### Mac Compatibility:

- **Symptom:** Binary dependency installation failures
- **Cause:** Apple Silicon architecture conflicts
- **Solution:** Use API mode (`--asr-provider gemini --tts-provider google`)

### Performance Issues:

- **Symptom:** Slow processing on Apple Silicon
- **Cause:** CPU-only processing
- **Solution:** Ensure MPS device detection (`--device mps`)

## Debug Modes

### Timeline Analysis:

```
vtbp translate video.mp4 output.mp4 --vad-mode --keep-temp
# Check work/segments_silero.json for VAD timing
```

### API Cost Estimation:

```
vtbp translate video.mp4 output.mp4 --estimate-cost --asr-provider gemini
```

### Credential Validation:

```
vtbp translate --validate-apis --asr-provider gemini --tts-provider google
```

---



# Future Roadmap

---

## Planned Enhancements

### Technical Improvements:

- WebRTC integration for real-time processing
- GPU-accelerated audio mixing
- Advanced voice cloning integration
- Multi-speaker diarization support

### User Experience:

- Web-based GUI interface
- Batch video processing
- Advanced timing controls
- Real-time preview capabilities

### Quality Enhancements:

- Lip-sync optimization
- Emotion preservation in translation
- Regional accent support
- Professional dubbing workflows

---

## Conclusion

VTBP represents a sophisticated approach to AI-powered video dubbing, combining cutting-edge machine learning models with professional audio engineering principles. The hybrid architecture allows users to optimize for quality, cost, or compatibility while maintaining consistently high output standards.

The VAD-based processing mode represents a significant innovation in timing accuracy, while the API integration approach solves platform compatibility issues that have historically limited accessibility to high-quality dubbing tools.

**Key Achievement:** Eliminated the traditional trade-off between quality and compatibility by leveraging cloud AI services for computationally intensive tasks while maintaining local control over audio engineering and final output quality.

---

*This documentation covers VTBP version 2.0 with API integration and VAD-based processing capabilities. For implementation details, see the source code modules referenced throughout this document.*