

Optimizing Voice Assistant Latency Architectures for Edge-Constrained CPU and Hybrid Environments: A Comprehensive Analysis

1. Executive Summary

The pursuit of fluid, naturalistic Human-Computer Interaction (HCI) through Voice User Interfaces (VUIs) is fundamentally a battle against latency. In the context of the "ATLAS" voice pipeline, the current latency of 6–15 seconds represents a catastrophic friction point that degrades the user experience from conversational to transactional. Psychological research in HCI suggests that the human threshold for "conversational" response times lies between 200ms (for backchanneling) and 700ms (for turn-taking). Latencies exceeding 2 seconds break the illusion of presence and cognitive flow.

This report presents an exhaustive engineering analysis and optimization strategy for the ATLAS system, operating within the specific constraints of 16GB RAM and an RTX 3050 Ti (4GB VRAM), primarily relying on CPU inference. The core thesis of this analysis challenges the assumption that the 4GB GPU is "unusable." While insufficient for local execution of frontier-class Large Language Models (LLMs), the RTX 3050 Ti is mathematically sufficient and architecturally critical for offloading the peripheral components of the pipeline—specifically Automatic Speech Recognition (ASR/STT) and Text-to-Speech (TTS).

The proposed optimization strategy targets a reduction of total system latency to under 2 seconds (P50) by implementing a Hybrid-Compute architecture. This approach reserves the CPU for asynchronous orchestration and LLM API management while leveraging the GPU for tensor-heavy signal processing. Key recommendations include the transition to faster-whisper with aggressive beam search pruning, the implementation of Anthropic's Prompt Caching to minimize Time-to-First-Token (TTFT), the deployment of Kokoro-82M via ONNX Runtime on the GPU, and the adoption of a fully asynchronous, streaming-first pipeline architecture.

2. Hardware Constraints and Architectural Analysis

The optimization of any high-performance AI system begins with a rigorous accounting of computational resources. In the case of ATLAS, the constraints are defined by a consumer-grade edge environment. Understanding the precise bottlenecks of this

architecture—specifically the interplay between CPU instruction sets, memory bandwidth, and GPU memory limitations—is prerequisite to effective optimization.

2.1 The Misconception of "Unusable" VRAM

The initial project context asserts that the RTX 3050 Ti’s 4GB VRAM is "unusable for LLMs." This statement, while accurate regarding the local execution of state-of-the-art 7B+ parameter models (which typically require 6–8GB VRAM for 4-bit quantization plus KV cache overhead), erroneously conflates LLM requirements with the requirements of the entire pipeline. In a modular voice architecture, the LLM is merely one of three distinct deep learning models.

The peripheral models—Whisper (STT) and Kokoro (TTS)—have significantly lower memory footprints. An analysis of the parameter counts and quantization profiles reveals a different reality:

- **Whisper Small (STT):** The "small" variant of OpenAI's Whisper contains approximately 244 million parameters. When loaded in FP16 (half-precision), this model requires approximately 500MB of VRAM. Using INT8 quantization, this requirement drops further, often below 400MB, depending on the implementation overhead.¹
- **Kokoro-82M (TTS):** As its name implies, Kokoro utilizes 82 million parameters. In standard FP32 precision, this equates to roughly 330MB of memory. In quantized ONNX formats (e.g., q8 or fp16), the runtime footprint can be as low as 150MB.²

Table 1: Theoretical VRAM Budgeting for Hybrid-Compute

Component	Precision	Estimated VRAM Usage	Status
OS / Display	N/A	~400 - 600 MB	Mandatory Overhead
STT (Whisper Small)	INT8 / FP16	~500 MB	Feasible
TTS (Kokoro-82M)	FP16 / ONNX	~300 MB	Feasible
PyTorch/CUDA Context	N/A	~400 MB	Overhead
Total Usage		~1.6 - 1.8 GB	

Remaining Headroom		~2.2 GB	Available
--------------------	--	---------	-----------

The data indicates that the RTX 3050 Ti is not only usable but is the single most critical asset for reducing latency. By offloading STT and TTS to the GPU, we free the CPU from heavy matrix multiplication tasks, reducing contention with the Python Global Interpreter Lock (GIL) and allowing the CPU to focus on network I/O and orchestration logic. This Hybrid-Compute approach—GPU for signal processing, CPU for logic and API handling—is the foundational architectural shift required to meet the <4 second target.³

2.2 Analysis of CPU-Only Bottlenecks

Current implementations of voice assistants on CPU-only architectures suffer from specific, predictable bottlenecks. If the system is forced to run strictly on the CPU (e.g., due to driver conflicts or headless server deployment), optimization becomes a game of instruction set efficiency and thread scheduling.

SIMD Utilization and Instruction Sets: Modern x86 CPUs (Intel Core i7/i9 or AMD Ryzen) utilize Single Instruction, Multiple Data (SIMD) extensions like AVX2 or AVX-512 to accelerate mathematical operations. However, Python-based deep learning frameworks often incur significant overhead moving data between Python objects and the underlying C++ execution backends (like CTranslate2 or ONNX Runtime). While faster-whisper optimizes this via CTranslate2, the CPU must still perform context switching between the heavy compute of transcription and the real-time requirements of audio capture.

Memory Bandwidth Saturation: With 16GB of system RAM, capacity is rarely the issue; bandwidth is. During inference, large weight matrices must be moved from RAM to the CPU registers. Unlike GPUs, which feature high-bandwidth GDDR6 memory, CPUs are limited by the system's DDR4/DDR5 speeds. This results in "memory-bound" inference where the compute units wait for data, introducing latency spikes, particularly when multiple models (STT and TTS) attempt to access memory simultaneously during turn-taking.⁵

Thread Contention: A monolithic Python script handling VAD, STT, and TTS on the CPU will suffer from the OS scheduler fighting for resources. The audio input stream (PyAudio/SoundDevice) requires real-time priority to avoid buffer overruns (glitches). If a heavy inference task utilizes OMP_NUM_THREADS equal to the total core count, the audio capture thread may starve, causing dropouts or delays in detecting the end of speech.

3. The Acoustic Front-End: Speech-to-Text (STT)

Optimization

The current pipeline reports a 2–4 second latency for STT using faster-whisper small on CPU. This single stage consumes the entire latency budget of an idealized voice assistant. Reducing this to under 500ms is imperative.

3.1 Model Selection and Architecture Comparison

The landscape of open-source ASR has shifted dramatically with the release of Whisper, but for latency-critical applications, the vanilla implementation is insufficient.

3.1.1 Faster-Whisper vs. OpenAI Whisper

The transition from the original OpenAI Whisper implementation to faster-whisper is the correct initial step. The original OpenAI library is built on PyTorch, which is designed for research flexibility rather than inference speed. faster-whisper is a reimplementation using **CTranslate2**, a custom inference engine for Transformer models that implements aggressive memory optimization and reduced-precision inference.⁶

Benchmarks consistently demonstrate that faster-whisper is up to 4 times faster than the original PyTorch implementation on the same hardware due to CTranslate2's efficient memory access patterns and weight quantization handling.¹ However, the configuration of this engine is where significant gains remain unlocked.

3.1.2 Distil-Whisper: The Efficiency Frontier

For further gains, the pipeline should transition to **Distil-Whisper**. These models are knowledge-distilled versions of the large-v3 model, trained to mimic the output of the larger model while possessing significantly fewer layers.

- **Distil-Small.en:** This model retains much of the robustness of the larger models regarding background noise and accents but significantly reduces parameter count and inference time.
- **Latency Impact:** Distil-Whisper models can offer a 50% reduction in latency compared to their non-distilled counterparts with minimal impact on Word Error Rate (WER) for clear, near-field speech typical of voice assistant interactions.¹

3.1.3 Alternative Models: Vosk, Canary, and Wav2Vec2

The research query asks about alternatives like Vosk, Canary, and wav2vec2.

- **Vosk (Kaldi-based):** Vosk is extremely fast on CPU (often <100ms) because it uses older HMM/GMM or lightweight neural architectures. However, it lacks the semantic robustness of Transformers. It is brittle to noise and requires specific language packs that may not handle modern vernacular or code-switching well. It is a valid fallback for extremely low-power devices (Raspberry Pi Zero) but is likely an unnecessary quality regression for

a machine with 16GB RAM.⁷

- **NVIDIA Canary:** Canary is a powerful multi-lingual model but is significantly heavier and optimized for NVIDIA server-grade GPUs. On a 3050 Ti, the overhead of managing its large context window may outweigh accuracy gains.
- **Wav2Vec2:** While efficient, Wav2Vec2 often requires an external language model (n-gram) for high accuracy decoding, adding complexity to the pipeline. Whisper's sequence-to-sequence nature handles punctuation and capitalization natively, which is critical for the downstream LLM to understand sentence structure.

Conclusion: Stick with Whisper, but optimized variants. `distil-whisper-small.en` via `faster-whisper` is the optimal balance of speed and intelligence.

3.2 Deep Dive: CPU Optimization Parameters

When running on CPU, the specific decoding parameters passed to the transcribe function have massive implications for computational cost.

3.2.1 Beam Size Pruning

The default `beam_size` in many Whisper implementations is 5. This means the model maintains 5 parallel hypotheses during the auto-regressive decoding process, constantly evaluating the top-5 most likely next tokens.

- **Computational Cost:** Running beam search is computationally expensive, scaling roughly linearly with the beam size.
- **Optimization:** Reducing `beam_size` to 1 (also known as Greedy Decoding) drastically reduces CPU load.
- **Data:** Research indicates that reducing beam size from 5 to 1 can improve inference speed by 2-3x on CPU. While this theoretically increases the risk of hallucination or repetition loops, for short command-style utterances, the accuracy drop is negligible.⁸

3.2.2 Quantization (INT8 vs FP32)

On modern x86 CPUs, INT8 quantization is essential. `faster-whisper` supports `compute_type="int8"`, which utilizes vector neural network instructions (VNNI) on supported CPUs (AVX2/AVX-512). This not only reduces the memory bandwidth requirement—often the choke point for CPU inference—but also doubles the theoretical throughput of the compute units.⁸

3.2.3 Threading and Core Affinity

A common pitfall is allowing the inference engine to utilize all available threads. If `faster-whisper` consumes 100% of CPU cycles, the Python interpreter (which manages the VAD loop and audio buffer) may stutter, leading to dropped audio frames.

- **Recommendation:** Explicitly set `cpu_threads` to the number of *physical* cores, leaving

1-2 cores free for the OS and audio capture. Do not use logical cores (Hyperthreading) for matrix math, as the shared cache resources often lead to performance degradation.⁸

3.3 Voice Activity Detection (VAD) Tuning

VAD is the gatekeeper of latency. The "Turn-Taking Threshold Paradox" describes the tension between interrupting the user (cutting them off mid-thought) and lagging (waiting too long to confirm silence).¹⁰

3.3.1 Silero VAD vs. WebRTC

The current pipeline likely uses a standard VAD. Upgrading to **Silero VAD** is recommended. Silero is a neural network-based VAD that is significantly more robust to background noise than energy-based methods like WebRTC. It prevents false positives from fans, typing, or distant noise, ensuring the STT engine is only triggered by actual speech.¹¹

3.3.2 Latency Tuning Parameters

The default silence timeout in many assistants is 1000ms (1 second). This adds a mandatory 1-second delay to every interaction.

- **Optimized Configuration:**
 - `min_silence_duration_ms`: **300ms to 400ms**. Research into human conversational dynamics suggests that gaps longer than ~500ms usually signal a turn exchange. Setting this to 350ms feels "snappy" without being overly aggressive.¹²
 - `speech_pad_ms`: **30ms**. Adding a small padding ensures the end of the last word isn't clipped.
 - `threshold`: **0.4 - 0.5**. A slightly lower threshold allows for softer endings to sentences.¹⁴

3.3.3 VAD Streaming Architecture

Instead of recording a file and then processing it, VAD should be applied to the audio stream in real-time chunks (e.g., 512 samples or 30ms).

- **Buffer Logic:** The system accumulates audio frames in a ring buffer. The VAD evaluates the latest chunk. If `is_speech` transitions to false for the duration of `min_silence_duration_ms`, the buffer is immediately flushed to the STT engine. This decoupling allows the STT model to be "warmed up" or allows the system to pre-load the context.¹⁵

3.4 GPU Offload Strategy

As established in Section 2.1, the RTX 3050 Ti is the preferred execution provider.

- **Implementation:** `model = WhisperModel("small.en", device="cuda", compute_type="int8")`.

- **Performance Delta:** A 13-minute audio file that takes ~195s on CPU (OpenAI Whisper) or ~18s on CPU (Faster-Whisper) can be processed in mere seconds on a GPU. For short utterances (3-5 seconds), GPU inference is effectively instantaneous (<200ms), largely eliminating STT as a bottleneck.¹
-

4. The Cognitive Core: LLM Latency Reduction

The current pipeline uses the Claude API via CLI, with a reported latency of 3–8 seconds. This is the largest variable in the pipeline, encompassing network round-trips, queue times, and token generation.

4.1 Model Selection: Time-to-First-Token (TTFT)

In a voice context, the total generation time is secondary to the **Time-to-First-Token (TTFT)**. The moment the first token arrives, the TTS system can theoretically begin processing (see Section 5). Therefore, minimizing TTFT is the priority.

4.1.1 Claude 3.5 Haiku

Anthropic's Claude 3.5 Haiku is the optimal choice for this application.

- **Benchmarks:** Independent benchmarks show Haiku achieves a TTFT of approximately **0.36 seconds**, compared to Sonnet's **0.64 seconds** and Opus's significantly higher latency.¹⁷
- **Throughput:** Haiku generates approximately 65 tokens per second. Since average human speech is roughly 3-4 words (4-5 tokens) per second, Haiku generates text nearly 15x faster than it can be spoken. This buffer ensures that once the TTS starts, it will never "starve" or pause waiting for more text.¹⁸

4.2 Prompt Caching (Anthropic)

Voice assistants typically carry a heavy "System Prompt"—a large block of text defining the persona, available tools, and behavioral constraints (e.g., "You are ATLAS... your responses should be concise... you have access to Home Assistant..."). Sending this context with every request forces the API to re-process these tokens, adding latency and cost.

Anthropic Prompt Caching allows the API to store the pre-computed attention states of the prompt prefix.¹⁹

- **Mechanism:** By marking the system prompt as "cached" (via `cache_control`), the model skips the processing phase for that segment in subsequent turns.
- **Impact:** This can reduce the "Prompt Processing" latency by up to **85%** for long contexts. For a voice assistant maintaining a conversation history, this is a massive optimization.²¹
- **Economic Benefit:** Cached tokens are billed at a significantly reduced rate (\$0.30/M vs

\$3.00/M), reducing the operational cost of the assistant by up to 90%.²²

4.3 Semantic Caching (Local "Shortcuts")

A robust analysis of voice assistant usage reveals a power law distribution: a small number of queries ("Turn on lights," "What time is it?", "Stop") account for a disproportionate volume of traffic. Relying on an LLM for these deterministic queries is inefficient.

Semantic Caching involves using a local vector database to short-circuit the LLM API.²³

- **Implementation:**
 1. The user's speech is transcribed: "Turn on the kitchen lights."
 2. An embedding model (e.g., a lightweight all-MiniLM-L6-v2 running locally on CPU) converts this text into a vector.
 3. The system queries a local vector store (e.g., FAISS, ChromaDB, or GPTCache).
 4. If a semantically similar query (Similarity > 0.90) is found, the system returns the stored action or response immediately.
- **Latency:** A local vector search takes <10ms. The total time from STT to Response is reduced to ~50ms, effectively zero latency compared to the API call.²⁴
- **Tooling:** Libraries like gptcache provide out-of-the-box integration for this pattern, handling the embedding generation and similarity search logic.²⁴

4.4 Speculative Decoding and Predictive Handoff

While true speculative decoding (using a small draft model to guide a large model) is typically a server-side optimization, a client-side variant can be applied: **Predictive Regex**.

- For extremely high-frequency, low-variance commands, simple regex matching on the STT output is superior to any LLM.
- *Example:* If STT output matches `r"turn (on|off) (.*)"`, execute the Home Assistant hook immediately and synthesize a generic acknowledgement ("Okay") while the LLM is arguably still "thinking." This creates an illusion of instant responsiveness.

5. Vocal Synthesis: Text-to-Speech (TTS) Optimization

The current TTS implementation (Kokoro ONNX) takes 1–3 seconds per sentence. This is the final barrier to a conversational experience. If the LLM responds instantly but the user waits 3 seconds for audio, the system feels sluggish.

5.1 Kokoro-82M Architecture

Kokoro is an 82-million parameter model based on StyleTTS2 architecture. Its small size makes it uniquely suited for edge deployment compared to massive 1B+ parameter models like

MetaVoice or 500M+ models like XTTS-v2.²⁷

5.2 Hardware Acceleration: ONNX Runtime

The key to unlocking low latency with Kokoro is optimizing the **ONNX Runtime** execution provider.

- **GPU Execution:** Even though the RTX 3050 Ti is "unusable" for LLMs, it is a powerhouse for a model of Kokoro's size. By utilizing the `CUDAExecutionProvider` in ONNX Runtime, inference speeds can reach **25x to 50x real-time**.⁴ This means generating 1 second of audio takes roughly 20-40 milliseconds.
- **CPU Fallback:** If forced to use CPU, ensure the use of `kokoro-quant.onnx`. This 8-bit quantized version reduces the model size to ~80MB and allows for significant speedups on consumer CPUs compared to the FP32 version.² Benchmarks suggest CPU inference can achieve ~11x real-time speed, which is acceptable but less robust than GPU acceleration.⁴

5.3 Streaming Synthesis and Sentence Boundary Detection

Waiting for the full text response to render is inefficient. The system must implement **streaming synthesis**.

- **Mechanism:** The LLM streams tokens. The orchestrator accumulates these tokens until a sentence boundary is detected (punctuation marks like `., ?, !, ;`).
- **Action:** As soon as a boundary is hit, that sentence is dispatched to the TTS engine.
- **Parallelism:** While the first sentence is being spoken to the user, the LLM is generating the second sentence, and the TTS engine is processing it in the background. This "waterfall" effect masks the generation latency of all subsequent text.³⁰
- **Phonemization:** Kokoro requires a text-to-phoneme step (often using `espeak-ng` or `misaki`).³¹ This is a CPU-bound string processing task. It is critical to ensure this runs in a separate thread so it does not block the main audio output loop.

5.4 Alternatives Comparison

- **Piper:** Highly optimized C++ TTS. Very fast on CPU (Raspberry Pi capable). The voices are less emotive than Kokoro but extremely stable. Good fallback.
- **Edge-TTS:** Relies on Microsoft's cloud API. While high quality, it introduces network latency and privacy dependency. Not recommended for a low-latency local-first pipeline.
- **Coqui XTTS:** Significantly heavier (requires ~2GB+ VRAM just for TTS) and slower. Harder to fit alongside Whisper on a 4GB card.

Conclusion: Kokoro-82M on ONNX/CUDA is the optimal choice for quality/speed balance on the 3050 Ti.

6. Pipeline Parallelization and Orchestration

Moving from a sequential script (record() -> transcribe() -> query() -> speak()) to an asynchronous event loop is the single most effective architectural change.

6.1 Asynchronous Event Loop (Python asyncio)

The core application must be an asyncio loop handling concurrent streams. Python's asyncio is ideal for this, as the workload is a mix of I/O-bound tasks (Network/Audio) and CPU/GPU-bound tasks (Inference).

Proposed Architecture:

1. **Input Worker (The Ear):** Continuously reads the microphone buffer. VAD runs on these chunks. When silence is confirmed, it pushes the audio_buffer to the STT_Queue.
2. **STT Worker (The Scribe):** Awaits the STT_Queue. Runs faster-whisper on the GPU. Pushes the resulting text to the LLM_Queue.
3. **LLM Worker (The Brain):** Awaits the LLM_Queue. Initiates a streaming request to Anthropic (with Prompt Caching). It accumulates tokens and pushes complete_sentences to the TTS_Queue.
4. **TTS Worker (The Voice):** Awaits the TTS_Queue. Runs session.run() on the GPU. Pushes audio_bytes to the Playback_Queue.
5. **Playback Worker (The Mouth):** Awaits the Playback_Queue. Writes data to the sounddevice.OutputStream.

6.2 Pre-warming and Memory Management

Deep learning models, particularly those using ONNX or CTranslate2, incur initialization overhead (loading weights to VRAM, JIT compilation of CUDA kernels) during their first inference pass.

- **Startup Routine:** Upon application boot, the system should run a "dummy" inference on both STT and TTS models (e.g., transcribe 1 second of silence, synthesize the word "Ready").⁵ This moves the 2-3 second initialization penalty to the startup phase, ensuring the first user interaction is snappy.

6.3 Barge-In (Interruption) Logic

True conversational agents allow the user to interrupt. If the VAD detects speech while the system is speaking (Playback Worker is active), an interrupt event must fire.

- **Action:**
 1. sounddevice.stop() is called immediately to kill audio output.³²
 2. The LLM_Task is cancelled (stopping network data).
 3. The TTS_Queue and Playback_Queue are cleared.
 4. The system resets to listening state.

This requires careful state management to ensure the system doesn't hear its own voice and trigger a loop (Acoustic Echo Cancellation is helpful here, but software interruption logic is the first line of defense).³³

7. Latency Measurement and Tracking

You cannot optimize what you cannot measure. A feeling of "sluggishness" must be converted into millisecond-precision data.

7.1 Instrumentation Strategy

Using a Python decorator pattern to log the execution time of every stage is best practice. This creates a trace of every interaction.

Metrics to Track:

- **VAD Latency:** Time from physical end-of-speech to STT start.
- **STT Latency:** Duration of the transcription process.
- **LLM TTFT:** Time from API request to first token.
- **TTS Chunk Latency:** Time from receiving text to producing audio bytes.
- **E2E Latency:** The "Glass-to-Glass" equivalent: Time from VAD Trigger -> First Audio Sample Output.

7.2 Structured Logging (JSONL)

Logs should be machine-readable for trend analysis. The `python-json-logger` library enables creating `metrics.jsonl` files.³⁴

JSON

```
{"timestamp": "2026-01-04T12:00:01", "event": "turn_complete", "metrics": {"stt_ms": 450, "llm_ttft_ms": 600, "tts_first_chunk_ms": 250, "e2e_ms": 1300}}
```

7.3 Latency Targets (P50 vs P95)

- **P50 (Median):** The goal for a typical interaction (short command, fast network). Target: **1.5 seconds**.
 - **P95 (Tail):** The worst case (long query, network jitter, noisy audio). Target: **3.5 seconds**.
-

8. Implementation Strategy and Code Patterns

The following section provides concrete Python code structures to implement the architectural recommendations.

8.1 Dependencies

Bash

```
pip install faster-whisper kokoro-onnx sounddevice numpy anthropic asyncio loguru
pip install onnxruntime-gpu # Critical for NVIDIA acceleration
```

8.2 Logging Decorator Pattern

This decorator provides the instrumentation layer discussed in Section 7.

Python

```
import time
import functools
import json
from loguru import logger

def track_latency(stage_name):
    def decorator(func):
        @functools.wraps(func)
        async def wrapper(*args, **kwargs):
            start = time.perf_counter()
            result = await func(*args, **kwargs)
            duration = time.perf_counter() - start

            log_entry = {
                "stage": stage_name,
                "duration_ms": round(duration * 1000, 2),
                "timestamp": time.time()
            }
            # Log to JSONL file
```

```

        with open("latency_metrics.jsonl", "a") as f:
            f.write(json.dumps(log_entry) + "\n")
        return result
    return wrapper
return decorator

```

8.3 Asynchronous Pipeline Skeleton

This code demonstrates the "waterfall" parallelization and queue management.

Python

```

import asyncio
import numpy as np
import sounddevice as sd
from faster_whisper import WhisperModel
from kokoro_onnx import Kokoro
from anthropic import AsyncAnthropic

class VoicePipeline:
    def __init__(self):
        # 1. Pre-warm Models (Hybrid Compute)
        # STT on GPU (INT8)
        logger.info("Loading STT on GPU...")
        self.stt = WhisperModel("small.en", device="cuda", compute_type="int8")

        # TTS on GPU (CUDA Execution Provider)
        logger.info("Loading TTS on GPU...")
        self.tts = Kokoro("kokoro.onnx", "voices.json")
        # Note: Ideally, pass providers= to underlying session

        self.llm = AsyncAnthropic()

        # Queues for inter-task communication
        self.audio_queue = asyncio.Queue()
        self.text_queue = asyncio.Queue()
        self.audio_out_queue = asyncio.Queue()

        # Interruption Flag
        self.interrupt_event = asyncio.Event()

```

```

@track_latency("stt_inference")
async def process_stt(self):
    while True:
        audio = await self.audio_queue.get()
        # Run blocking inference in a thread to avoid freezing the loop
        segments, _ = await asyncio.to_thread(
            self.stt.transcribe,
            audio,
            beam_size=1,
            vad_filter=True,
            min_silence_duration_ms=350
        )
        text = " ".join([s.text for s in segments]).strip()
        if text:
            logger.info(f"User: {text}")
            self.interrupt_event.clear()
            asyncio.create_task(self.process_llm(text))

```

```

@track_latency("llm_stream")
async def process_llm(self, query):
    # Check Semantic Cache here (omitted for brevity)

```

```

    async with self.llm.messages.stream(
        max_tokens=1024,
        messages=[{"role": "user", "content": query}],
        model="claude-3-5-haiku-latest",
        system={"text": "You are ATLAS.", "cache_control": {"type": "ephemeral"}} # Prompt Caching
    ) as stream:
        sentence_buffer = ""
        async for text in stream.text_stream:
            if self.interrupt_event.is_set(): break
            sentence_buffer += text
            # Simple heuristic: Flush on punctuation
            if text in [".", "?", "!", ";"]:
                await self.text_queue.put(sentence_buffer)
                sentence_buffer = ""

```

```

        # Flush any remaining text
        if sentence_buffer and not self.interrupt_event.is_set():
            await self.text_queue.put(sentence_buffer)

```

```

@track_latency("tts_synthesis")

```

```

async def process_tts(self):
    while True:
        text = await self.text_queue.get()
        if self.interrupt_event.is_set(): continue

        # Offload to GPU via thread
        audio, sample_rate = await asyncio.to_thread(
            self.tts.create,
            text,
            voice="af_bella",
            speed=1.0
        )
        await self.audio_out_queue.put((audio, sample_rate))

    async def play_audio(self):
        while True:
            audio, rate = await self.audio_out_queue.get()
            if self.interrupt_event.is_set(): continue

            sd.play(audio, samplerate=rate)
            sd.wait() # Blocking wait; in production use callback for better interrupt handling

    async def run(self):
        # Initialize tasks
        await asyncio.gather(
            self.process_stt(),
            self.process_tts(),
            self.play_audio()
        )
        # Add VAD/Microphone capture loop here

    )

if __name__ == "__main__":
    pipeline = VoicePipeline()
    asyncio.run(pipeline.run())

```

9. Conclusion

The transformation of the ATLAS pipeline from a 15-second latency architecture to a sub-2-second conversational agent is technically achievable within the existing hardware footprint. The key insight lies in recognizing that the RTX 3050 Ti is perfectly sized for the acoustic models (Whisper and Kokoro), allowing the system to operate in a Hybrid-Compute

configuration.

By moving STT and TTS to the GPU, utilizing faster-whisper's CTranslate2 backend, and adopting Claude 3.5 Haiku with Prompt Caching, the system eliminates the primary compute and network bottlenecks. The shift to an asynchronous asyncio design ensures that these gains are realized in parallel, masking generation time behind playback.

Final Expected Latency Profile:

- **VAD/Capture:** 350ms
- **STT (GPU/Beam=1):** 400ms
- **LLM (Haiku TTFT):** 500ms
- **TTS (Kokoro GPU First Chunk):** 200ms
- **Total End-to-End: ~1.45 Seconds**

This architecture not only meets the <4 second requirement but pushes the system into the realm of true conversational fluidity. The next steps involve rigorous VAD parameter tuning and the collection of real-world latency logs to identify any remaining edge-case bottlenecks.

Works cited

1. Faster Whisper transcription with CTranslate2 - GitHub, accessed on January 4, 2026, <https://github.com/SYSTRAN/faster-whisper>
2. README.md · onnx-community/Kokoro-82M-ONNX at 6217b562792b37a8638668048dbfc505101c058f - Hugging Face, accessed on January 4, 2026, <https://huggingface.co/onnx-community/Kokoro-82M-ONNX/blob/6217b562792b37a8638668048dbfc505101c058f/README.md>
3. Would 2GB vs 4GB of VRAM Make Any Difference for Whisper? : r/speechtech - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/speechtech/comments/1k5qzsb/would_2gb_vs_4gb_of_vram_make_any_difference_for/
4. Introducing kokoro-onnx TTS : r/LocalLLaMA - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1htwkba/introducing_kokoroonnx_tts/
5. Whisper very slow on first access - Voice Assistant, accessed on January 4, 2026, <https://community.home-assistant.io/t/whisper-very-slow-on-first-access/599912>
6. Choosing between Whisper variants: faster-whisper, insanely-fast-whisper, WhisperX, accessed on January 4, 2026, <https://modal.com/blog/choosing-whisper-variants>
7. Build a Speech-to-Text Service in Python with Faster Whisper | by Johni Douglas Marangon, accessed on January 4, 2026, <https://medium.com/@johnidouglasmarangon/build-a-speech-to-text-service-in-python-with-faster-whisper-39ad3b1e2305>
8. With CPU, it's quite slow. A 2-minute MP3 takes 6 minutes to finish. · Issue #279 ·

- SYSTRAN/faster-whisper - GitHub, accessed on January 4, 2026,
<https://github.com/SYSTRAN/faster-whisper/issues/279>
9. All my attempts to improve accuracy and reduce hallucinations have the opposite effect!, accessed on January 4, 2026,
<https://community.openai.com/t/all-my-attempts-to-improve-accuracy-and-reduce-hallucinations-have-the-opposite-effect/997302>
 10. Voice Activity Detection (VAD) - Arun Baby, accessed on January 4, 2026,
<https://arunbaby.com/ai-agents/0019-voice-activity-detection/>
 11. Voice activity detection in text-to-speech: how real-time VAD works - QED42, accessed on January 4, 2026,
<https://www.qed42.com/insights/voice-activity-detection-in-text-to-speech-how-real-time-vad-works>
 12. Voice Activity Detection - A Lazy Data Science Guide - Mohit Mayank, accessed on January 4, 2026,
http://mohitmayank.com/a_lazy_data_science_guide/audio_intelligence/voice_activity_detection/
 13. speech package - github.com/skypro1111/silero-vad-go/speech - Go Packages, accessed on January 4, 2026,
<https://pkg.go.dev/github.com/skypro1111/silero-vad-go/speech>
 14. Quality Metrics · snakers4/silero-vad Wiki - GitHub, accessed on January 4, 2026,
<https://github.com/snakers4/silero-vad/wiki/Quality-Metrics>
 15. Real-time Discord STT Bot using Multiprocessing & Faster-Whisper : r/Python - Reddit, accessed on January 4, 2026,
https://www.reddit.com/r/Python/comments/1p211nn/realtime_discord_stt_bot_using_multiprocessing/
 16. silero-vad/src/silero_vad/utils_vad.py at master - GitHub, accessed on January 4, 2026,
https://github.com/snakers4/silero-vad/blob/master/src/silero_vad/utils_vad.py
 17. Claude 3.5 Haiku vs. Sonnet: Speed or Power? A Comprehensive Comparison, accessed on January 4, 2026,
<https://www.keywordsai.co/blog/claude-3-5-sonnet-vs-claude-3-5-haiku>
 18. Understanding Different Claude Models: A Guide to Anthropic's AI, accessed on January 4, 2026,
<https://teamai.com/blog/large-language-models-llms/understanding-different-claude-models/>
 19. Slashing LLM Costs and Latencies with Prompt Caching - Hakkoda, accessed on January 4, 2026, <https://hakkoda.io/resources/prompt-caching/>
 20. Prompt caching - Claude Docs, accessed on January 4, 2026,
<https://platform.claude.com/docs/en/build-with-claude/prompt-caching>
 21. Prompt caching: 10x cheaper LLM tokens, but how? | ngrok blog, accessed on January 4, 2026, <https://ngrok.com/blog/prompt-caching>
 22. Anthropic Prompt Caching Cuts AI API Costs 90% | byteiota, accessed on January 4, 2026,
<https://byteiota.com/anthropic-prompt-caching-cuts-ai-api-costs-90/>
 23. Semantic Cache: How to Speed Up LLM and RAG Applications - Medium,

- accessed on January 4, 2026,
<https://medium.com/@svosh2/semantic-cache-how-to-speed-up-llm-and-rag-applications-79e74ce34d1d>
24. GPTCache : A Library for Creating Semantic Cache for LLM Queries — GPTCache, accessed on January 4, 2026, <https://gptcache.readthedocs.io/>
 25. Semantic caching cut our LLM costs by almost 50% and I feel stupid for not doing it sooner : r/LangChain - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/LangChain/comments/1pzn06m/semantic_caching_cut_our_llm_costs_by_almost_50/
 26. zilliztech/GPTCache: Semantic cache for LLMs. Fully integrated with LangChain and llama_index. - GitHub, accessed on January 4, 2026, <https://github.com/zilliztech/GPTCache>
 27. Kokoro-82M: The best TTS model in just 82 Million parameters | by Mehul Gupta - Medium, accessed on January 4, 2026, <https://medium.com/data-science-in-your-pocket/kokoro-82m-the-best-tts-model-in-just-82-million-parameters-512b4ba4f94c>
 28. yaoyunqqq | Kokoro-82M - Kaggle, accessed on January 4, 2026, <https://www.kaggle.com/models/yaoyunqqq/kokoro-82m>
 29. Kokoro-82M ONNX Runtime Inference Gradio Demo, accessed on January 4, 2026, <https://yakhyo.github.io/kokoro-onnx/>
 30. Voice AI quickstart | LiveKit docs, accessed on January 4, 2026, <https://docs.livekit.io/agents/start/voice-ai-quickstart/>
 31. Kokoro TTS 82M (How To Have It Process From GPU instead of CPU)? - Reddit, accessed on January 4, 2026, https://www.reddit.com/r/LocalLLaMA/comments/1o382o7/kokoro_tts_82m_how_to_have_it_process_from_gpu/
 32. Convenience Functions using NumPy Arrays — python-sounddevice, version 0.4.0, accessed on January 4, 2026, <https://python-sounddevice.readthedocs.io/en/0.4.0/api/convenience-functions.html>
 33. Interruption handling over FastAPIWebsocket not working when Bot speaking · Issue #2460, accessed on January 4, 2026, <https://github.com/pipeocat-ai/pipeocat/issues/2460>
 34. Quick Start - Python JSON Logger, accessed on January 4, 2026, <https://nhairs.github.io/python-json-logger/latest/quickstart/>