ChatGPT

Improving EN↔RU Real-Time Speech Translation Backend

Current MVP Pipeline Overview

The prototype system uses a **cascade of AI models** for live Russian–English speech translation. On the input side, an **ASR/AST model** (NVIDIA's *Canary* 1B) performs speech recognition and translation in one step [1] [2]. It receives 16 kHz mono audio from the client (resampled and buffered in ~20 ms chunks) and outputs translated text. The backend then uses **Silero TTS** to synthesize speech from the translated text, sending audio back to the client [3] [4]. The FastAPI server manages a WebSocket streaming interface, accumulating audio until end-of-utterance (via a voice-activity detection gate) triggers a translation+TTS cycle [5] [6]. This design is functional, but there is room to **boost accuracy and performance** at each stage.

Canary AST Model: Quality and Limitations

NVIDIA Canary-1b-v2 is a 1 billion-parameter *multilingual* speech model supporting 25 European languages (including Russian and English) [7]. It's a **state-of-the-art** model for automatic speech recognition **and** direct speech-to-text translation (AST) – in fact, it tops Hugging Face's leaderboard for multilingual ASR accuracy as of 2025 [8]. In the MVP, Canary is used in AST mode to transcribe and translate audio in one pass [9] [10]. This yields high-quality results for RU↔EN generally, with *accurate transcription and decent translation* in everyday scenarios. Canary-1b-v2 was trained on the massive Granary corpus (~1M hours) and achieves translation quality on par with much larger models [11]. It also outputs **punctuation and capitalization**, which improves readability and TTS prosody [12].

Despite its strengths, the current Canary AST setup has **notable limitations**:

- **Model Size and Speed:** At 1B parameters, Canary is heavy. Inference is optimized (Fast-Conformer backbone, etc.), and NVIDIA reports it runs *up to 10× faster* than some rival models [11]. Still, real-time use benefits from GPU acceleration. If running on CPU, latency would be high; on GPU (which the code likely uses by default), latency for an utterance is better but can spike for long audio. The MVP caps utterances at ~8 seconds [13] to manage latency and memory. This avoids extreme cases but still, processing a full 8 s audio chunk means the user waits until the end to get any translation. There is **no partial result streaming** implemented – Canary's `transcribe()` processes the whole chunk before returning [10]. This all-at-once approach simplifies translation but adds delay.

- **Preprocessing & Segmentation:** The audio pipeline feeds 16 kHz mono PCM to Canary (the model's expected format [14]). This is correct – Canary is trained on 16 kHz audio. However, the end-of-utterance detection might cause **segmentation issues**. The client uses an RMS energy gate (300 ms hangover) to decide when to flush audio [15] [16]. If a speaker pauses briefly (e.g. mid-sentence), the gate could prematurely flush, leading Canary to translate an **incomplete sentence**. The model itself

is not optimized for *simultaneous* or prefix translation – it expects complete utterances. Indeed, NVIDIA notes that accuracy drops for *incomplete sentences or word-by-word input*, since context is missing [17]. In practice, this can yield awkward or partial translations if the utterance was cut short. The MVP tries to mitigate this by suppressing very short or repetitive outputs (e.g. just "yes/да") [18] [19], but this is a blunt heuristic. Important content could still be truncated.

- **Language Handling:** Canary supports English→Russian and Russian→English AST within one model [20]. The code correctly specifies `source_lang` and `target_lang` for each request [21]. One limitation is that **language auto-detection** isn't used – the source language must be set in advance. If the conversation switches languages unexpectedly, the system might mis-translate. (NVIDIA's newer *Parakeet* model can auto-detect input language, but Canary requires explicit source language [22] [23].) For an internal system where users know which mode (RU→EN or EN→RU) they need, this is acceptable, but it reduces flexibility.

- **Accuracy and Edge Cases:** Even cutting-edge models aren't 100% accurate. Canary's word error rate is low on average (e.g. ~8–9% on multilingual test sets) [24], but heavy accents, fast speech, or noise can still cause recognition errors [17]. The Granary training gave it robustness to noise – WER only rises modestly down to 5 dB SNR, though it worsens at extreme noise levels [25]. For translation quality, Canary-1b-v2 achieves BLEU ~29–40 on benchmark AST tasks [26]. This is solid, but a specialized machine translation system might score higher on certain domains. Being an end-to-end model, Canary may sometimes **omit names or jargon**, or *hallucinate* minor filler words, since it's optimizing for overall meaning. The model card explicitly warns that accuracy varies with input domain, accent, and context [17]. In RU↔EN, a well-resourced pair, translation is generally good, but idioms or very colloquial speech might not always carry over perfectly.

In summary, *Canary AST provides a strong base* for the MVP with broad language support and integrated translation. The main challenges are **latency (processing whole utterances at once)** and potential errors with **incomplete or difficult inputs**. We'll address improvements like streaming decoding and alternative ASR/NMT options in upcoming sections.

Silero TTS Performance and Voice Quality

For speech synthesis, the MVP uses the **Silero TTS** library with pre-trained voices (Russian *v4_ru* "aidar" and English *v3_en* "en_0") [27]. Silero models are lightweight FastSpeech2-type architectures known for **fast, intelligible speech**. In this setup, Silero runs on CPU with a 24 kHz output sample rate [28] [29]. The generated audio is then resampled to 16 kHz for streaming to the client [30]. This TTS component is relatively easy to use and has a small in-memory cache to avoid reloading models repeatedly [31] [32].

Voice quality: Silero TTS produces clear, easy-to-understand speech. Evaluations of Russian TTS show that FastSpeech2-based models like Silero achieve the **best clarity** – phonemes are articulated accurately with minimal distortion [33] [34]. In Nickolay Shmyrev's 2024 benchmarks, Silero's Russian voices had the lowest character error rates (CER ~0.7) among open-source models, outperforming even some big-tech cloud TTS in clarity [35] [34]. Users will notice that Silero's output is **crisp and natural in pronunciation**, with no odd buzzing or muffled sounds. This makes the translated speech easy to comprehend. However, like many FastSpeech2 systems, Silero can sound somewhat **flat or robotic in prosody**. Its intonation and expressiveness aren't as rich as human speech – sentences tend to have a uniform cadence. As the

evaluation noted, these models have *"plain intonation but clarity is hard to beat"* [34]. For an internal translation tool, this trade-off (high clarity vs. moderate expressiveness) is usually acceptable. It ensures the translation is heard clearly, even if the tone is a bit monotonous or pauses aren't as human-like.

Performance bottlenecks: Silero TTS is designed for efficiency. The developers report that Silero v3/v4 can run *"blazingly fast even on one CPU thread"* [36]. Indeed, in the above evaluation, Silero v4 achieved a real-time factor of ~0.05–0.13× on CPU (meaning it can synthesize speech **20× faster than realtime** on modern hardware) [37]. In our MVP, typical translated sentences (~5–10 words) synthesize in well under a second on CPU. So the TTS model itself is usually *not* the slowest link. That said, a few aspects could be impacting performance or latency in the current integration:

- **File I/O Overhead:** The code writes the generated audio to a WAV file (`tempfile`) and then reads it back into memory [38]. This ensures compatibility with the Silero API (which outputs to file by default), but the disk I/O introduces some overhead. For each utterance, writing and reading a WAV file (even in a tempfs or RAM disk) is inefficient. It incurs a small delay proportional to audio length and system I/O speed. While short sentences produce small files (few hundred KB), eliminating this step could shave off tens of milliseconds and reduce CPU load slightly.

- **Single-Threaded CPU Use:** By default, PyTorch TTS runs on CPU using multiple threads (it will utilize vectorization and parallelism for mel generation and vocoder). The Silero library likely uses PyTorch under the hood; if not constrained, it might spawn threads. The MVP doesn't explicitly configure `torch.set_num_threads()`, so it's presumably using the default. If the host machine has many cores, TTS may be able to use them (thus being very fast). However, if the CPU is under heavy load or if threads are limited, TTS could become a bottleneck for longer text. Running TTS on the GPU is an option (the Silero `device` parameter could be set to `"cuda"` [39] [29]), which might cut synthesis time further, especially for longer outputs, at the cost of some VRAM. In the MVP, the choice to use CPU might be to avoid GPU context-switch overhead or because the GPU is fully occupied by the 1B AST model. We should examine this trade-off under **optimizations**.

- **Voice and Model Versions:** The system hard-codes one English voice (en_0) and one Russian voice (aidar). These are decent neutral voices. Silero v4_ru actually offers multiple speakers (male and female) [40]. If the current voice is deemed too flat or not ideal, switching to another pretrained speaker (e.g., "baya" or "xenia" for a female voice) is trivial and might subjectively improve user experience. As for model versions, **v4_ru** is the latest Russian model (good choice), and **v3_en** is the latest English model (there is no v4_en as of the release). Upgrading English TTS to a newer release (if/when Silero releases v4 English) could improve quality or speed. For instance, Silero's Russian v4 model halved CPU latency compared to v3 in benchmarks [37] – a similar gain might come for English with a new version. It's worth keeping an eye on Silero's updates for a potential drop-in improvement.

In summary, Silero TTS in the MVP is already **high-performing and clear**, with minor areas to fine-tune. The main quality issue is the somewhat robotic intonation, which might be acceptable for an internal tool. Next, we outline concrete steps to **improve recognition accuracy, translation fidelity, and overall speed** by tweaking or replacing components.

Improving ASR and Translation Quality

To boost **audio recognition accuracy and translation quality** for Russian↔English, we can pursue two parallel strategies: (1) **maximize Canary's performance** through tuning and data handling, and (2) consider a **modular ASR + NMT pipeline** with specialized models for potentially higher quality. We also ensure the text output is well-formed for TTS.

1. Tune and Utilize Canary AST Fully: The current Canary integration is minimal – we load the model and call `transcribe` with source/target languages. A few adjustments could improve its output:

- *Enable Punctuation/Capitalization:* Canary can output punctuated, capitalized text (PnC). The code appears to strip tokens like `<|pnc|>` and `<|en|>` from the raw result [41] [42]. This suggests the model might be producing control tokens rather than actual punctuation, possibly because we didn't explicitly request "pnc=yes" in the input. In NeMo, when using manifest files, one can set `"pnc": "yes"` to get punctuated text [43] [44]. For direct API calls, we should ensure the model is configured to include punctuation (or use the newer Canary checkpoint which may default to PnC). Having proper punctuation will make translations more readable and can help TTS insert natural pauses (e.g. at commas or end of sentence). NVIDIA notes that Canary provides *"accurate punctuation and capitalization"* in its output [12], so we should take advantage of this feature. If necessary, we can run a lightweight post-processor (like a punctuation restoration model [45]) on Canary's output text, but ideally Canary itself can handle it.

- *Adjust Decoding Parameters:* By default, Canary might use a **beam search** for transcription. The model card for Canary-1b (earlier version) shows examples setting `beam_size = 1` for faster decoding [46] [47]. We should verify if Canary-1b-v2 is using beam search; if so, consider using **greedy decoding** (beam size 1) to reduce latency. A beam search can marginally improve accuracy at the cost of significant computation per utterance. In real-time use, a small drop in BLEU/WER might be acceptable for quicker responses. Since Canary is a high-quality model, even greedy decoding likely yields good results. We can make this configurable – for critical use, use beam=4; for everyday use, beam=1 for speed. Additionally, if we observe any systematic errors (e.g., certain words frequently misrecognized), NeMo's adaptation features like **decoding biasing** could be explored. For example, if internal jargon or names are important, we might supply a custom bias token list to help the model recognize those out-of-vocabulary terms. (This is more common in ASR than AST, but Canary's ASR backbone might support it.)

- *Long Utterances and Context:* If users sometimes speak longer than the 8 s cap, we should handle that carefully. Canary-1b-v2 supports **dynamic chunking** internally for long-form audio (splitting input with overlaps) [48]. This is great for efficiency, but extremely long inputs (minutes) might still strain GPU memory. Since our design flushes at 8 s or on pause, we likely won't hit those limits. However, we might consider a **slightly longer hangover** in VAD to avoid chopping in the middle of a clause. For instance, increasing the silence hangover from 300 ms to ~500 ms could reduce accidental mid-sentence flushes. It's a trade-off: too long a hangover adds post-utterance delay. We can also monitor the *content* of partial transcripts – if the user is clearly in the middle of a sentence (conjunctions like "and", unfinished grammar), maybe wait a bit more before finalizing. In practice, a 0.5 sec hangover often suffices to catch a speaker's brief pause.

- *Accent and Noise Robustness:* For accuracy under diverse conditions, we can employ front-end audio enhancements. The client already can enable noise suppression and echo cancellation via WebRTC APIs [49] [50]. We should encourage those settings for better input quality (they can significantly boost ASR accuracy in noisy environments). Additionally, using a high-quality VAD (like Silero VAD [51]) could better detect speech segments, though the current simple RMS gate is working and lightweight. If certain accents or dialectal speech are common internally, we might explore fine-tuning Canary on in-domain data – but that is a heavy process (requiring training on more transcripts). Given Canary's broad training, it likely handles most standard Russian and English accents decently.

2. Cascade ASR + NMT Alternative: Another path to improve translation quality is to split the problem into two specialized tasks: **speech recognition** and **text translation**. This "cascade" approach sacrifices the end-to-end nature of Canary in favor of possibly higher accuracy in each component. Specifically:

- *Use a High-Accuracy ASR:* For Russian and English speech, we could use models like **OpenAI's Whisper** for transcription. Whisper (Large-v2) is a 1.5B-parameter model trained on 680k hours of multilingual data, known for robust transcription in many languages. It achieves very low WER on both English and Russian. For example, Whisper Large's English WER ~2.7% on LibriSpeech, and it handles Russian with strong accuracy (OpenAI reported ~10.5 BLEU on ru→en speech translation via Whisper, but that included translation step) – Canary claims to beat Whisper, but Whisper is still a strong baseline. If we use Whisper, we'd do: **RU speech → RU text**, **EN speech → EN text** (Whisper can also directly translate non-English speech to English text in "translate" mode, but it does not output Russian). So for a symmetric system, we'd use it purely for transcription in the source language. Another candidate is **NVIDIA Parakeet** model (0.6B) for ASR [52]. Parakeet is optimized for speed and can *auto-detect the language*, outputting transcripts in the original language [53]. It has slightly lower accuracy than Canary (being smaller), but much higher throughput. For instance, Parakeet can transcribe extremely long audio in one pass and topped HF's throughput charts [52]. A Parakeet-based ASR might be useful if we need to handle faster streams or multiple streams concurrently. The choice boils down to required accuracy vs. speed.

- *Use a Strong Machine Translation (NMT):* Once we have source-language text, we can translate it with a dedicated NMT model. For **English→Russian** and **Russian→English**, there are many proven solutions. For instance, Facebook's WMT19 English-Russian model (based on FAIR's Transformer) is available open-source and achieved near human-level BLEU in the WMT competition. Helsinki-NLP's OPUS-MT models for en-ru are also readily available (small but decent). More ambitiously, Meta AI's **NLLB-200** model (No Language Left Behind) includes high-quality RU↔EN translation; the 1.3B parameter version could be used if we have the compute (it's a text model that could run on GPU or CPU with INT8 quantization). These text translators are trained on massive bilingual corpora (news, web text, etc.), so they might handle colloquial phrases or domain-specific terms better than Canary's end-to-end model. For example, if the internal use involves technical jargon, an NMT model fine-tuned on that domain's text would likely translate more accurately than a generic AST model.

- *Benefits and Trade-offs:* The cascade approach potentially **improves translation fidelity** – each component can be best-in-class. It also offers **flexibility**: the same English transcript could be translated to multiple languages if needed, by swapping the NMT model, without re-running ASR. Moreover, we could get intermediate transcripts (which might be useful for debugging or for showing the original speech text to bilingual users). On the downside, errors can compound (ASR

mistakes will be carried into translation). However, modern ASR like Whisper makes very few mistakes on clear input. Even if Canary's direct AST is slightly better integrated (e.g., it might correct an ASR error in the translation stage), the difference may be minor. Another trade-off is **latency**: doing ASR then MT sequentially could be slower than Canary's single pass. But we can mitigate this. Whisper transcribes in near real-time on a GPU (especially if we use a medium-sized model for speed). NMT on a single sentence is usually very fast (millisecond-range on GPU, or a few hundred ms on CPU for a big model). We could also overlap the processes: e.g., start translating the first part of the transcript while Whisper is finishing the last part (though handling reordering in translation is complex – better to wait for full sentence). For typical sentence-length utterances, the extra delay might be small.

- *Implementability:* We can integrate Whisper (or another ASR) via its Python API or through NeMo (NVIDIA has NeMo ASR models for Russian – e.g. QuartzNet or Conformer models – though Whisper tends to be more accurate). For NMT, there are Hugging Face Transformers pipelines or the Marian NMT library. For instance, using Hugging Face's `Facebook/wmt19-en-ru` model with `torch.half()` on GPU could translate a sentence in ~50 ms. We would need to manage two models instead of one, and ensure proper text normalization (Whisper adds punctuation and casing in transcripts, which actually helps NMT). Whisper would give us punctuated text by default, or we could still use Canary for ASR mode (itself is a great ASR). In fact, a *hybrid approach* could be: Canary/Parakeet for **transcription only** (source_lang == target_lang, so it does ASR with punctuation), then feed that text into a separate translator. That leverages NVIDIA's model for recognition (which is strong and already in use) and a specialized NMT for translation. It would avoid any regression in ASR quality from switching models, and likely improve translation quality especially for tricky sentence constructions. Keep in mind, Canary's internal translation was trained on speech data which might be shorter or simpler sentences on average; a text MT model trained on news articles, etc., might handle complex syntax or idioms more gracefully.

- *Use of Large Language Models:* (For completeness, since this is an internal system.) One could use a large language model (LLM) like GPT-4 or similar to do translation with very high quality, preserving nuance. For instance, transcribe with ASR then prompt an LLM to translate. However, that introduces dependency on external APIs (if using OpenAI) or the overhead of running a huge model locally, so it's likely not practical here. But it's worth noting that some companies do integrate cloud translation for quality. Given our priority on performance and on-premise use, sticking to open models is preferable.

In summary, switching to a cascade ASR+NMT pipeline could yield **incremental accuracy gains** in translation at the cost of added complexity. It's a viable approach if Canary's direct AST occasionally produces unsatisfactory translations. We should empirically compare a few translations from Canary vs. a strong text MT to decide. If Canary's output is already acceptable for our use, we might stick with it and instead focus on the latency and throughput improvements discussed next.

Reducing Latency and Improving Throughput

To make the system more **responsive and scalable**, we can refine the backend architecture and use of resources. Key suggestions include enabling *streaming*, parallelizing where possible, caching smartly, and optimizing hardware utilization:

- **Implement Incremental Streaming:** Currently, users get the translated speech only after they finish an utterance and the system processes it. This can create a noticeable pause for longer sentences. By adopting streaming, we could provide *partial transcripts or audio* sooner. There are two levels to consider:
- *ASR partial results:* We can leverage Canary (or an alternate ASR) to produce interim text while the user is still speaking. For example, if 3 seconds of speech have been received, we could run a recognition on that chunk to get a preliminary transcript, and update it as more audio comes in. This is tricky with direct AST (since a partial translation of a half sentence might be misleading). One approach: perform streaming **ASR in the source language** to display live captions (or just to have them internally), and only commit to a translation when a sentence is finished. This is how some interpreter systems work – they show original speech text in real time but wait for a clause to complete to output translated speech. For an internal tool, showing live transcription (even untranslated) could be useful feedback. If translation needs to be streamed too, one could attempt incremental translation: e.g., translate each clause as it comes. However, with languages like RU↔EN, word order differences mean a naive left-to-right translation can be inaccurate until the sentence is finished. True simultaneous translation is an active research area. Unless we have a model specifically designed for it, it might be safer to only stream the source transcript and wait for end-of-sentence for the final translation.

- *Chunked TTS output:* If the output speech is long, we can stream audio in chunks. For instance, as soon as we have the first part of the translated text, start synthesizing and playing it while the rest is being processed. The MVP already sends the whole TTS audio once ready. But we could cut very long outputs into sentences or phrases and pipeline the synthesis. This would require the TTS to produce seamlessly concatenable audio chunks (which is feasible if we split on punctuation). Given typical usage (likely short conversational sentences), this may not be a priority optimization – it's more useful if translating something like a long paragraph or continuous speech.

- **Parallelize ASR and TTS on GPU/CPU:** We should take advantage of available hardware by running tasks in parallel where possible. For example, if we have a single GPU, we might run the Canary AST on it, while simultaneously running TTS on the CPU. In the current sequential design, TTS starts only after AST finishes [3]. But since those two tasks use different hardware, we could overlap them in time. One way: once Canary returns text, immediately launch TTS in a background thread *and* meanwhile start listening to the next user utterance (i.e., unpause the audio capturing). With careful timing, the system could be "listening" for the next input while still speaking out the translation of the previous input. This full-duplex behavior would make conversations more fluid. The client UI would need to handle the case of user talking over the TTS playback if that's a concern (maybe by auto-pausing the mic while TTS plays, etc., depending on use case). Even without full duplex, we can at least ensure the CPU and GPU are utilized concurrently. Right now, after AST (GPU) completes, the GPU sits idle during the CPU TTS synthesis. If we enabled **GPU TTS**, we could instead have the GPU do both jobs back-to-back; that's not parallel, but GPU might do TTS faster than CPU. However, running both AST and TTS on the GPU *concurrently* is only possible if we had two GPUs or if the

models are small enough to share one GPU with asynchronous execution (Canary 1B likely uses most of a GPU's compute while running). So the more practical parallelism is **GPU for AST, CPU for TTS** – which is what we have, but done sequentially. We can pipeline it better by overlapping operations (e.g., perform audio resampling and formatting on one thread while TTS is synthesizing on another, etc.). Using Python `asyncio` or background threads for the heavy calls could ensure the WebSocket loop isn't stalled completely during processing [54].

- **Leverage Multi-Core and Batch Processing:** If the internal use might involve *multiple concurrent translations* (say multiple users or a meeting setting with two streams), we should scale horizontally. We could run separate instances of the pipeline per user (each on a different GPU or CPU core). Since FastAPI and the WebSocket can handle concurrent connections, we'd want to avoid any global locks around the models. Loading one global Canary model and one global TTS instance is memory-efficient, but it means requests are essentially serialized through those (the code is not multi-threading calls to `canary_mgr.translate()`). NeMo's ASRModel is not inherently thread-safe for simultaneous use, and the GIL in Python would also limit true parallel execution. For better throughput, we could spawn worker processes that each own a model instance. For example, a small pool of worker processes (via Celery or FastAPI's background tasks) could handle incoming audio frames. This would use more memory (each process loads models), but would allow parallel inference on multi-core CPUs or multi-GPU servers. An alternative is to use **batching**: Canary's `transcribe` can accept a list of audio samples for batch inference [55] [47]. If we had multiple audio requests to translate at once, we could batch them in one forward pass (especially if using the same GPU). This reduces per-request overhead. However, in a low-volume internal setting, batching likely isn't needed – it's more beneficial when many requests pile up.

- **Caching Outputs:** We already cache the loaded models, which is good (no reinitialization cost each time). We could consider caching *frequent translation results*. For instance, if the system often hears the same phrase (e.g., "Hello, how are you?" might be common), we could cache the last N translations and their TTS audio. Then if the same exact audio/text comes again, we instantly reuse the cached translation/audio. This is a classic trade-off of memory vs. time. In a dynamic speech scenario, exact repeats aren't super common (except things like "yes", "no", which we are suppressing from TTS anyway). So this might have limited benefit. Still, for testing/demo, it could make sense (to avoid recomputation when trying the same input repeatedly). A more useful cache could be at the *text translation* level: if using a separate NMT, cache translation of recently seen sentences, since text exact matches are more likely to repeat in some contexts. Given this is an internal MVP, caching isn't critical unless we observe specific repetitive patterns.

- **Memory Optimization:** Running a 1B model and TTS can consume significant memory (GPU ~4 GB for Canary in FP16, CPU RAM for TTS maybe a few hundred MB). On a modern machine this is fine. But if we needed to deploy on a smaller device or share resources, we could use techniques like **model quantization**. For example, converting Canary to INT8 with minimal accuracy loss could cut GPU memory usage, and even allow running on lower-end GPUs. NVIDIA might release optimized FP8 or INT8 checkpoints; otherwise, one could try ONNX Runtime or TorchQuant utilities to quantize. This is advanced and might not be necessary given that performance, not memory, is the priority here. One memory-related improvement that *is* simple: **pre-load models at startup**. The MVP lazily loads Canary on first use [56], causing a big delay the first time a translation is requested. We can call `canary_mgr.load()` and `tts_mgr = TTSManager(...)` on application startup instead. This way, the first user interaction doesn't incur model loading time. This is especially important if the

system will be started and stopped during demos or tests – having it "ready to go" improves perceived performance. The cost is a slightly longer app startup time and higher idle memory use, which is usually acceptable for an always-on internal service.

In essence, by introducing **streaming partial results**, better **concurrency**, and some micro-optimizations (removing file I/O, caching where appropriate, pre-loading models), we can significantly reduce the end-to-end latency. For example, a user could finish a sentence and start hearing the translation **almost immediately (within ~200 ms)** if we apply these improvements, rather than perhaps 1–2 seconds currently (for model + TTS processing). This will make conversations feel more natural. Throughput (the ability to handle multiple translations per minute or multiple users) will also increase by ensuring we use all available CPU/GPU resources effectively [52].

Specific Recommendations Summary

To conclude, here are the **actionable steps** drawn from the above analysis:

- **Upgrade/Tune the Models:**
- Use **Canary-1b-v2** in AST mode with punctuation enabled – ensure we get formatted text (or apply a post-process) so TTS has nicer input [12].
- Consider **separate ASR + NMT**: e.g. Canary or Whisper for raw transcription, then a strong text translator (Facebook WMT19 or OPUS-MT) for RU↔EN. This could improve translation accuracy for complex sentences. Evaluate this pipeline on real samples to see if the gain is worth the added complexity.
- Experiment with **NVIDIA Parakeet (0.6B)** for faster transcription if latency is paramount. Parakeet auto-detects language and is optimized for real-time, potentially useful for rapid prototyping [52] [53].

- Keep models updated: if NVIDIA releases a Canary v3 or if Silero releases English v4, plan to upgrade. These tend to bring quality or speed improvements (e.g., Canary v2 expanded languages and improved efficiency over v1 [11]).

- **Improve Recognition & Translation Quality:**

- Increase VAD hangover to avoid cutting off utterances mid-sentence (e.g. 500 ms vs 300 ms) – this gives the model more complete input, improving translation quality.
- Use **noise suppression and echo cancellation** on input (already available in WebRTC) and possibly add server-side noise filtering for robustness. Clean audio in = better ASR [25].
- If using cascade pipeline, apply text normalization to ASR output (ensure things like numbers, abbreviations are expanded properly for translation). Tools or rules can be applied so that "Dr." isn't read as "drive" etc., although Canary/Whisper often handle basic formatting.

- If certain errors are observed (e.g., names), consider a **custom dictionary or bias**. For instance, NeMo allows adding hints so ASR is more likely to recognize specific proper nouns [17]. Similarly, a glossary can be applied in some NMT frameworks to enforce certain translations. For an internal system with known terminology, this can boost accuracy for critical terms.

- **Optimize Silero TTS:**

- Remove the temp file write/read in `synth()` – instead modify the Silero TTS call to return a numpy array in memory (the Silero library might allow generating to a buffer or we can intercept the audio before it's written to disk). This will cut redundant I/O and reduce latency a bit.
- **GPU-accelerate TTS** if latency for long outputs becomes an issue. Silero on GPU can synthesize in ~0.015× real-time [57], which is essentially instantaneous for normal sentences. The GPU has to load the model (a few hundred MB) – if memory allows, it can be worth it. In a scenario where the GPU is mostly idle (AST runs for e.g. 200 ms, then sits free), doing TTS on it utilizes it better. We should profile CPU vs GPU for typical utterances; given Silero's CPU speed, we might find CPU is fine. But it's good to have the GPU option, especially if we adopt heavier TTS in future or multi-lingual voices.

- Add support for **multiple voices** or style if desired. Silero supports many voices (e.g., 5 Russian voices) [40]. We could allow a configuration to pick a voice (male/female) for the output speech. This doesn't improve "accuracy" per se, but could improve user perception and make the translated speech sound more pleasant or appropriate for the context. For example, perhaps choose a Russian female voice if the English speaker was female, to better match the tone. This could be a simple toggle in the UI that maps to choosing a different `speaker` in the TTSManager.

- **Latency and Throughput Enhancements:**

- Enable **partial results streaming**. At minimum, show live transcription of the speaker's speech in the UI as they talk (this can use the same AST model in ASR mode or a separate ASR). This gives immediate feedback. Only play the translated audio once a full sentence is detected to avoid oscillations in translation.
- Pipeline operations to **overlap processing**. For instance, start capturing the next utterance while the translation audio for the previous utterance is still playing. The app can be configured to do so if full duplex is acceptable. Technically, since the playback audio is routed to an `<audio>` element (with potential echo cancel enabled) [50] [58], it might not be picked up by the mic anyway, but careful testing is needed to avoid feedback loops.
- Use Python concurrency: run the Canary translation in an `asyncio.to_thread` or background task so that the WebSocket loop can continue handling incoming audio chunks simultaneously. Likewise for TTS synthesis. This prevents any single long operation from blocking the reception of new audio. The code structure is already asynchronous, so integrating this should be feasible.
- For scalability, consider a **process-per-client** model. If multiple people will use the translator at once, spawning a separate worker (with its own models) avoids contention. This could be done via Docker containers or simply multiprocessing. Since this is internal, even a manual approach (launch one instance for each known user station) might be fine. If needing dynamic scaling, integrate a task queue where audio frames are forwarded to a GPU inference service. This is probably overkill for now, but keeping the architecture modular (separating the concerns of audio I/O vs. model inference) can ease future scaling.
- Monitor performance and adjust the **utterance length limit** if needed. The 8-second limit is a safeguard – if users often speak longer, increasing it (with the awareness of higher latency) might be warranted, or instructing users to speak in shorter segments might be necessary. Very long sentences can be hard to translate anyway (even for humans), so a balanced approach is best.

By implementing these recommendations, the system should see **notable improvements**. Empirically, we expect to reduce end-to-end latency (from voice input to translated voice output) and improve translation accuracy/fluency. For example, using a top-notch text MT may improve BLEU scores for RU↔EN by a few points compared to end-to-end [26], and proper punctuation will make the output sound more natural.

Likewise, streaming partial transcripts can cut perceived wait times by several seconds in long utterances. These changes are grounded in the capabilities of the models and frameworks as documented (NVIDIA and Silero benchmarks) and should be implementable without enormous effort – mostly configuration and integration work, rather than training new models from scratch. Overall, the backend will become **faster, more accurate, and more robust**, aligning with the goal of a high-performance internal speech translation tool.

Sources:

- NVIDIA Canary Model Card and Blog – model capabilities, performance, and supported languages [7] [8] [11]
- NVIDIA NeMo documentation – using AST vs. ASR, punctuation control, and performance notes [43] [48] [17]
- Silero TTS GitHub/wiki – model versions, speed benchmarks, and quality evaluation for Russian voices [35] [33]
- Code excerpts from the MVP (Canary integration and TTS usage) – current architecture and settings [21] [27] .

---

[1] [7] [17] [20] [24] [25] [26] [48] nvidia/canary-1b-v2 · Hugging Face
https://huggingface.co/nvidia/canary-1b-v2

[2] [3] [4] [5] [6] [13] [18] [19] [21] [30] [39] [54] [56] app.py
file://file-9geFbf3azeBFBrtm1BqZWr

[8] [11] [12] [22] [23] [52] [53] NVIDIA Releases Open Dataset, Models for Multilingual Speech AI | NVIDIA Blog
https://blogs.nvidia.com/blog/speech-ai-dataset-models/

[9] [10] [41] [42] canary_ast.py
file://file-JsHjqdZY8LMqKQu1g5i1AW

[14] [43] [44] [46] [47] [55] nvidia/canary-1b · Hugging Face
https://huggingface.co/nvidia/canary-1b

[15] [16] encoder.worklet.js
file://file-QbiQMdBCVYWiJDpzWQGU8N

[27] [28] [29] [31] [32] [38] silero_tts.py
file://file-S44Hod5pJJwhJp48wXppAU

[33] [34] [35] [37] [57] Evaluation of Russian TTS models | Speech Recognition With Vosk
https://alphacephei.com/nsh/2024/07/12/russian-tts.html

[36] [P] Silero TTS Full V3 Release : r/MachineLearning
https://www.reddit.com/r/MachineLearning/comments/v9rigf/p_silero_tts_full_v3_release/

[40] GitHub - snakers4/silero-models: Silero Models: pre-trained speech-to-text, text-to-speech and text-enhancement models made embarrassingly simple
https://github.com/snakers4/silero-models

[45] Punctuation and Capitalization Model — NVIDIA NeMo 1.0.0rc1 ...
https://docs.nvidia.com/deeplearning/nemo/archives/nemo-100rc1/user-guide/docs/nlp/punctuation_and_capitalization.html

49 50 58 app.js
file://file-5SyQMrKbKWQs79R7gmEZxG

51 How to Implement High-Speed Voice Recognition in Chatbot ...
https://medium.com/@aidenkoh/how-to-implement-high-speed-voice-recognition-in-chatbot-systems-with-whisperx-silero-vad-cdd45ea30904