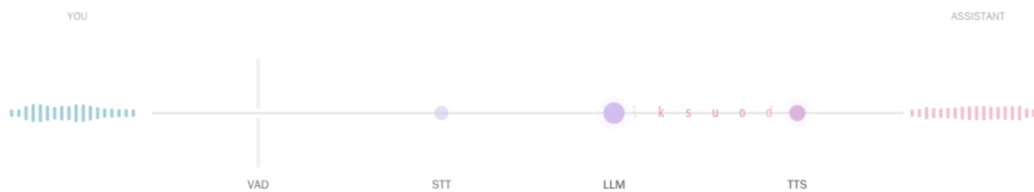
Hugging Voice

How we built our open realtime voice API.



Same client protocol. Swappable STT, LLM, and TTS. Local, hosted, or self-hosted.

AUTHORS

Andres Marafioti, Amir Mahla, Thibaud Frere,
Leandro Von Werra, Thomas Wolf

AFFILIATION

Hugging Face

PUBLISHED

Jul. 09, 2026

Table of Contents

1 1. One-line migration

2 2. How We Reverse-Engineered OpenAI's Realtime API

- 2.1 The protocol tells you what, never how
- 2.2 A deliberately minimal surface

3 3. Architecture: A Modular Cascade Behind a Realtime API

4 4. Latency Budget: Where Time Actually Goes

5 5. Choosing Your LLM / Provider

- 5.1 Hosted providers
- 5.2 Responses API or Chat Completions?
- 5.3 Fully local
- 5.4 In-process local backends

6 6. Tool Calling: One Wire Protocol, Multiple Backend Strategies

- 6.1 Tool calls are just tokens
- 6.2 Who does the parsing?
- 6.3 Code as the universal tool-calling language
- 6.4 The three backend paths, side by side
- 6.5 Closing the loop

7 7. Interruption and Barge-In

- 7.1 Generation-tagged output
- 7.2 What an interruption looks like

8 8. As Multilingual as the Mixture

9 9. What This Unlocks

10 Where this goes next

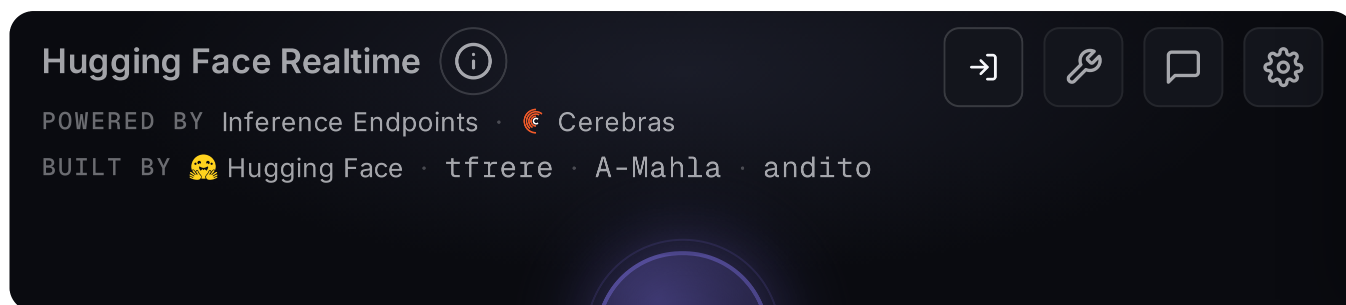
We've shipped 9,000 Reachy Minis with conversation capabilities. Together, they now generate 15k hours of conversation every month. On GPT-realtime, that would cost at least $45k a month. We know this because we used GPT-realtime on the first batch of robots back in January, and with just 2,000 units shipped, we spent $15k on it in a single month.

A normal conversation on GPT-realtime costs roughly $3/hour when set up correctly (caching matters a lot), and more when it isn't (background noise will rack up your bill fast!). We needed something we could serve at scale for cheap, without making the robot feel any less alive.

So we built our own 'gpt-realtime'. We reverse-engineered the OpenAI Realtime protocol, so existing clients connect with a single line change, but the backend is a fully open, end-to-end cascade we control: VAD, STT, LLM, and TTS, each swappable. Today, it costs us $0.25/hour to serve, and if you run it locally, which you can, it costs you nothing but electricity.

We're open-sourcing the whole stack. And we're not done: over the coming months, we plan to bring our serving cost down another 10x, to roughly $0.02/hour.

Here's how we built it, and where we're going.



1. One-line migration

OpenAI's realtime API is the go-to voice-agent API today. Which is why we used it when we were rushing to ship the reachy mini robots. To enable everyone to switch, we made our project compatible with OpenAI's. You can just change the URI and you are done :)

Hosted OpenAI realtime

```python
# client.py
from openai import OpenAI

client = OpenAI(
    base_url="https://api.openai.com/v1",
    api_key="sk-proj-xxxxxxxxxxxxxxxx",
)

# speech-to-speech, fully self-hosted
```

2. How We Reverse-Engineered OpenAI's Realtime API

Being OpenAI-compatible wasn't the plan. It became the plan.

Making our own engine speak the *exact same protocol* meant existing clients could switch backends with a one-line change, no rewrite, no migration guide, no risk. A few thousand Reachy Minis later, it was clearly the right call. But getting there was not as straightforward as "read the docs, implement the events."

The protocol tells you *what*, never *how*

The Realtime API is a bidirectional event protocol over a WebSocket at `/v1/realtime`. The client sends JSON events (`input_audio_buffer.append`, `session.update`, `response.create`, …), the server streams JSON events back (`session.created`, transcription deltas, audio deltas, `response.done`, …). Audio travels as base64-encoded PCM inside those events.

Here's the catch: the protocol is organized by *domain* (sessions, conversations, responses, audio buffers), not by system. It describes a contract between client and server, and says absolutely nothing about what sits behind it: nobody outside OpenAI knows whether their backend is a single end-to-end speech-to-speech model or a cascade, because the protocol

doesn't let you tell. Ours is a cascade of four models connected by queues. So for every event, we had to answer three questions:

1. What is the hidden logic behind this event? What does the client *expect* to happen, including all the implicit state transitions the docs don't spell out?
2. What is its reality in our architecture? Which of our components (VAD, STT, LLM, TTS, router) does it touch, and in what order?
3. What are the interconnections? Events are not independent: a `response.cancel` changes the meaning of every audio delta still in flight.

Take interruption as an example. In the protocol, barge-in looks simple: the server emits `input_audio_buffer.speech_started`, and the active response ends with `response.done`. Behind our cascade, that one event has to fan out into a whole choreography: cancel the LLM mid-stream, cancel the TTS mid-synthesis, flush every queue between them, and discard stale audio that's already been generated but not yet sent, all without ever violating the wire contract the client relies on. One event on the wire, five components to coordinate behind it. (We give this its own section below, because barge-in is where most voice demos fall apart.)

So we went baby step by baby step: implement one event, hook Reachy Mini up, watch where reality diverged from expectation, fix, repeat. This iterative loop, with a real robot, real conversations, and real interruptions, is what shaped the engine. And honestly, this work was never just plumbing. Mapping someone else's client-facing contract onto our own architecture forced trade-offs and design decisions that made the engine more robust and more *client-driven* than if we had invented our own protocol from scratch.

A deliberately minimal surface

The full Realtime protocol is big: the current `openai-python` SDK defines around 45 distinct event types, and the list keeps growing with every API update. We intentionally implemented the subset needed for a fully functional voice agent: audio in, turn detection, live transcription, streamed audio out, tool calling, and interruption. That's 5 client events and 13 server events:

Client → Server (5) Server → Client (13)

Event	What you're doing
`input_audio_buffer.append`	Stream microphone audio
`session.update`	Change instructions, tools, voice, or turn detection
`conversation.item.create`	Inject text or a tool result into the context (no generation triggered)
`response.create`	Ask for a response
`response.cancel`	Cancel the response mid-flight

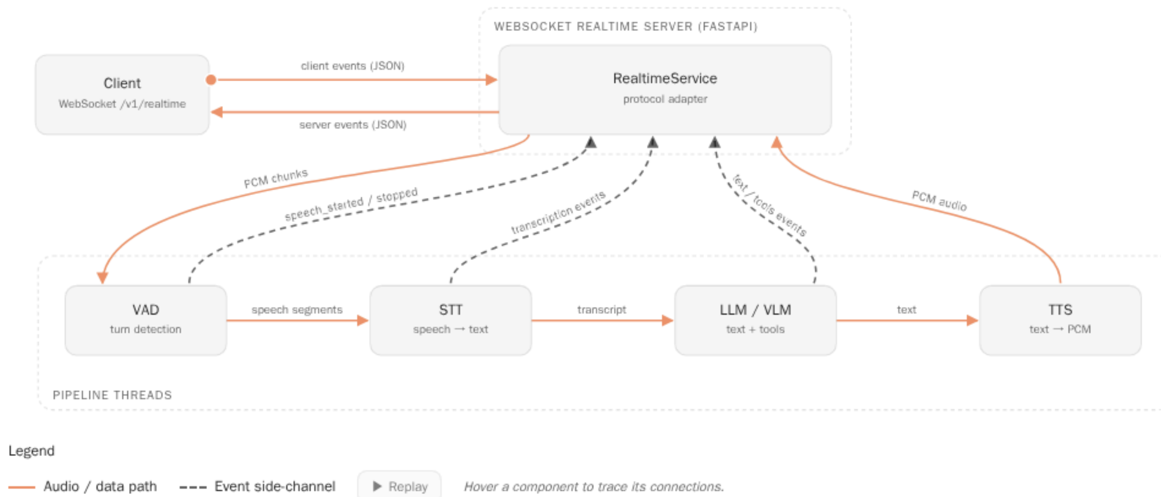
One event deserves a special mention: `response.function_call_arguments.done`. It's what separates a voice *chatbot* from a voice *agent*. Without it, the model can only talk; with it, the client receives structured tool calls it can execute (move the robot's head, search the web, check a camera) and feed results back into the conversation. We cover how three very different LLM backends all converge onto this single event in the tool-calling section.

Everything else in the protocol (`conversation.item.truncate`, `rate_limits.updated`, out-of-band responses, DTMF events) is not implemented yet, and that's on purpose. The event surface is open, the architecture is designed for it, and we'd love the community's help covering the rest of the protocol with us. If your client needs an event we don't emit yet, that's a great first PR. :)

3. Architecture: A Modular Cascade Behind a Realtime API

The most important design choice is that OpenAI compatibility lives at the edge. The public surface is a WebSocket at `/v1/realtime`, but behind that endpoint the server is a queue-driven cascade: audio comes in, speech is detected, text is transcribed, the LLM generates text and tool calls, TTS turns selected text back into PCM, and the router translates everything back into OpenAI-style realtime events.

In realtime mode, `speech-to-speech` starts a FastAPI/uvicorn server and creates a pool of isolated `PipelineUnit`s. Each connected client claims one unit. That unit owns its queues, events, cancellation state, chat state, and handler chain, so two clients do not share in-flight audio or response state.



Incoming audio is handled by the protocol layer first: the server decodes the base64 PCM, resamples it to the pipeline rate, cuts it into 512-sample chunks, and puts them on the VAD input queue together with the current runtime config. From there, every stage is just a handler reading one queue and writing the next one.

The audio path is linear, but the event path is not. The stages also feed a side event channel for everything the client must know before, during, or after audio playback: speech starting and stopping, partial and final transcripts, assistant text, token usage, errors, and tool calls. The async send loop drains both the audio queue and this event queue, then emits the actual realtime protocol events, synthesized audio included.

`RealtimeService` is the adapter in the middle. It parses client events, updates session state, appends final transcripts and tool results into chat context, triggers generation requests, and maps internal pipeline events back to the OpenAI Realtime event schema. It also holds a mutable session config shared with every handler: `session.update` can change instructions, tools, voice, audio formats, and turn detection settings mid-conversation without changing the client protocol.

That separation is what makes the cascade useful. VAD can tune turn detection without touching TTS. STT can stream partial transcripts without forcing the LLM to wait for every UI event. The LLM backend can be OpenAI-compatible, local Transformers, MLX, or something behind llama.cpp, while the client still sees the same `/v1/realtime` contract. TTS can optimize chunking, warmup, or voice selection independently, because it only receives clean text plus runtime config and returns PCM.

So the project is not a single voice model pretending to be an API. It is a protocol adapter over a set of replaceable realtime components, with queues between them and one canonical event layer around them. That gives us three practical knobs: scale by adding pipeline units, swap models by changing handlers, and debug latency by looking at the boundaries between stages.

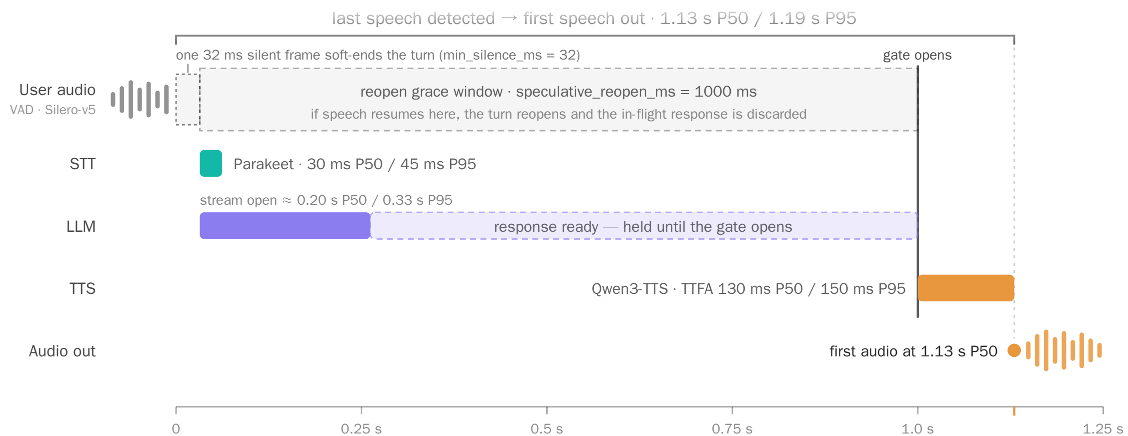
4. Latency Budget: Where Time Actually Goes

We care about the latency between the user's last detected speech and the first playable assistant audio. All numbers in this section are measured from our production logs on that path.

We resample inbound audio to 16 kHz and process 32 ms frames. The first process in the pipeline is the Voice Activity Detection (VAD), for which we use Silero-v5. We detect speech after 384 ms of active audio. This makes the system resilient to noise, but it also makes it deaf to very short utterances like "Ok". After the VAD detects speech, we yield it every 0.5 s for progressive transcriptions, which can be used for robots to quickly react to the user's speech. We don't mind doing STT so often because we rely on Parakeet, which is very fast: about 30 ms P50 and 45 ms P95, even with P95 utterances running 15 seconds of audio. For utterances longer than 15 seconds, we divide them into sentences and stop continuously transcribing sentences that were already done.

The first interesting part is what happens at the end of a turn: how much silence should the VAD wait for before deciding the user is done? This deployment waits for as little as possible: `--min_silence_ms 32`, a single 32 ms frame. One silent frame and the turn "soft-ends": it's provisionally over, pending confirmation. That would normally be absurdly trigger-happy, but it works because of speculative turn handling: on soft-end, the pipeline immediately dispatches STT and then the LLM request, while VAD opens a reopen grace window (`speculative_reopen_ms`, 1000 ms by default). If the user resumes speaking inside that window, the turn reopens and the in-flight response is discarded. If they don't, the response is released to the TTS.

With a slow LLM you get this behavior almost for free: the user's continuation arrives before the response does, so there is nothing to hold back. With a model as fast as the one we serve on Cerebras, the response would land 200 ms after a single silent frame, so we explicitly hold the output until the window expires. The stream-open latency to Cerebras (our closest TTFB proxy) was about 0.20 s P50 and 0.33 s P95.



Anatomy of the median turn, measured from the last detected speech frame. A single 32 ms silent frame soft-ends the turn; STT and the LLM then run speculatively inside the reopen window, so TTS is the only stage paying latency after the gate: 1.00 s + 0.13 s ≈ 1.13 s P50.

Because we wait to close the speaker turn, the effective endpointing delay is ~1 s, but the models do their work during it, not after it.

TTS is the only stage that runs fully after the gate. Our implementation of Qwen3-TTS logs TTFA at 130 ms P50 and 150 ms P95 live. Generation runs around 4.8x realtime, comfortably ahead of playback. This also determines the concurrency we can accept per GPU. 4 users fit comfortably in one instance, because even if all users are generating speech at the same time, our realtime factor handles it well. We can even go higher, but we'll start to see delayed speech on the higher percentiles.

As we said at the start, the most important metric is "last speech detected to first speech out." In our deployed system, that is 1.13 s P50 and 1.19 s P95. The median decomposes almost exactly as grace window + TTS TTFA: 1.0 s + 0.13 s. Everything else (STT, LLM first output) fits inside the gate and contributes nothing at the median. That also identifies the dominant median-latency knob: `speculative_reopen_ms`. Shrinking it buys latency directly, but it is a turn-taking tradeoff, not a performance one. Bringing it lower makes the system interrupt the user!

The distribution being nearly flat from P50 to P95 is not an accident: a timer-paced pipeline should look like this. The remaining tail (P99 1.65 s, max 2.95 s) comes from turns where the LLM takes longer than the gate, like tool calls and long contexts.

5. Choosing Your LLM / Provider

The LLM is the slowest, most expensive part of the cascade. VAD runs in microseconds, STT and TTS are small local models, but a single forward pass through the language model can dominate both your end-to-end response time and your bill. So this is the component you'll want to swap the most, and it's the one we made the easiest to swap.

The key abstraction: the realtime voice layer never changes. Your client still speaks the OpenAI Realtime protocol, VAD still detects turns, STT and TTS still run locally. Only the text-in, text-out box in the middle changes. The LLM slot speaks OpenAI-compatible protocols on the *other* side too, so anything that exposes `/v1/responses` or `/v1/chat/completions` plugs straight in: a hosted provider, HF Inference Providers, OpenRouter, or a vLLM / llama.cpp server on your own hardware. You can prototype against a hosted frontier model, then move to a self-hosted open model for production, without touching a single line of the voice pipeline or your client.

Here's the decision matrix:

Goal	`--llm_backend`	Where the LLM runs	
Fastest hosted setup	`chat-completions`	provider	Gemma 4 3
Default hosted setup	`responses-api`	provider	OpenAI, HF
Self-hosted server	`responses-api` or `chat-completions`	cloud or local	vLLM, llam
Apple-maxxing	`mlx-lm`	local	MLX model
CUDA-maxxing	`transformers`	local	HF model,

Hosted providers

The default backend is `responses-api`, which targets `/v1/responses`. Point `--responses_api_base_url` at your provider and set `--model_name`:

Provider / server	`--responses_api_base_url`	`--responses_api_api_key`
OpenAI	omit (uses OpenAI default)	`$OPENAI_API_KEY`
HF Inference Providers	`https://router.huggingface.co/v1`	`$HF_TOKEN`
OpenRouter	`https://openrouter.ai/api/v1`	`$OPENROUTER_API_KEY`
vLLM	`http://localhost:8000/v1`	omit or any string
llama.cpp	`http://127.0.0.1:8080/v1`	empty string

For example, running Qwen3.5-9B through HF Inference Providers, served by Together:

```
speech-to-speech \
    --llm_backend responses-api \
    --model_name "Qwen/Qwen3.5-9B:together" \
    --responses_api_base_url "https://router.huggingface.co/v1" \
    --responses_api_api_key "$HF_TOKEN" \
    --responses_api_stream
```

`--responses_api_stream` makes the backend consume tokens as the model generates them instead of waiting for the full completion. Every example here includes it, because voice needs the first sentence the moment it exists. The `model:provider` suffix is an HF router feature: same URL, same token, and you can hop between providers (`:together`, `:groq`, `:cerebras`, …) just by editing the model name. For voice, this matters more than for chat: first-token latency varies wildly between providers serving the same model, and you feel every millisecond of it as dead air before the assistant starts talking.

Responses API or Chat Completions?

Both API backends share the same `--responses_api_*` connection flags, so switching between them is a one-flag change:

- `--llm_backend responses-api` (default) targets `/v1/responses`.
- `--llm_backend chat-completions` targets `/v1/chat/completions`.

Prefer `chat-completions` in two situations we've hit in practice:

1. The provider ignores reasoning controls on the Responses path. Reasoning tokens are poison for voice latency: the user hears silence while the model "thinks." Some providers

only respect `reasoning_effort` on Chat Completions (see #312). Add `--responses_api_reasoning_effort none` to turn it off:

```
# Gemma 4 31B on Cerebras via the HF router, reasoning disabled for low voice latency
speech-to-speech \
    --llm_backend chat-completions \
    --model_name "google/gemma-4-31B-it:cerebras" \
    --responses_api_base_url "https://router.huggingface.co/v1" \
    --responses_api_api_key "$HF_TOKEN" \
    --responses_api_reasoning_effort none \
    --responses_api_stream
```

2. Streaming tool calls are flaky on the server's Responses path but solid on Chat Completions. We've seen this with some vLLM builds. Same server, same model. Just talk to the endpoint that streams tool calls reliably (more on how tool calls flow through the pipeline in the tool calling section below):

```
# vLLM serving a Qwen model with tool calling
speech-to-speech \
    --llm_backend chat-completions \
    --model_name "Qwen/Qwen3-4B-Instruct-2507" \
    --responses_api_base_url "http://localhost:8000/v1" \
    --responses_api_stream
```

Fully local

The lowest-friction fully local setup keeps the LLM in a separate llama.cpp process. This is exactly how the local Reachy Mini conversation works:

```
# Terminal 1: llama.cpp serving Gemma 4
llama-server -hf ggml-org/gemma-4-E4B-it-GGUF -np 2 -c 65536 -fa on --swa-full

# Terminal 2: speech-to-speech pointing at it
speech-to-speech \
    --llm_backend responses-api \
    --model_name "ggml-org/gemma-4-E4B-it-GGUF" \
    --responses_api_base_url "http://127.0.0.1:8080/v1" \
    --responses_api_api_key "" \
    --responses_api_stream
```

To the pipeline, llama.cpp is just another "provider." Nothing about the voice layer knows or cares that the model is running three centimeters away instead of in a datacenter.

In-process local backends

If you'd rather skip the separate server, two backends load the model directly into the pipeline process:

- `mlx-lm` on Apple Silicon. This is what `--local_mac_optimal_settings` picks, and combined with Parakeet TDT for STT and the MLX-quantized Qwen3-TTS, it turns a MacBook into a self-contained voice agent:

```
speech-to-speech \
    --local_mac_optimal_settings \
    --model_name mlx-community/Qwen3-4B-Instruct-2507-bf16
```

- `transformers` on CUDA or CPU, for any text-generation model on the Hub:

```
speech-to-speech \
    --llm_backend transformers \
    --model_name google/gemma-2b-it
```

Whichever path you pick, the wire protocol your client sees is identical: same events, same transcripts, same tool calls. That's the point. Choosing an LLM should be a deployment decision, not a rewrite.

6. Tool Calling: One Wire Protocol, Multiple Backend Strategies

A voice assistant that can only talk is just that: an assistant. One that can *do things*, like check the weather, move a robot's head, or look something up, is an agent. Tool calling is what separates the two, and it's also where the "swap any LLM backend" promise gets hard.

On the wire, tool calling in our server always looks the same, no matter which LLM sits behind it:

1. The client declares its tools through `session.update` (standard JSON Schema, same as OpenAI Realtime).

2. The LLM backend, whichever one you picked, produces tool calls in its own way.
3. The server normalizes everything into `response.function_call_arguments.done` events, with a `call_id`, a `name`, and JSON `arguments`.

Step 2 is where the fun is. To understand why, we need to talk about how tool calling actually happens inside an LLM generation.

Tool calls are just tokens

There is no “function calling mode” inside a language model. When a model calls a tool, it’s doing the only thing it knows how to do: predicting the next token. Tool calling is a *convention*: the model is trained to emit its function calls inside special delimiter tokens, and the chat template injects the tool definitions into the prompt so the model knows what’s available.

You can see this directly with Transformers (this snippet comes straight from the tool calling section of the Transformers docs):

```
inputs = tokenizer.apply_chat_template(
    messages, tools=tools, add_generation_prompt=True,
    return_dict=True, return_tensors="pt"
)
outputs = model.generate(**inputs.to(model.device), max_new_tokens=128)
print(tokenizer.decode(outputs[0][len(inputs["input_ids"][0]):]))
```

Run this with a Qwen model and a weather tool, and the raw generation looks like this:

```
<tool_call>
{"arguments": {"location": "Paris, France", "unit": "celsius"}, "name": "get_current_temperature"}
</tool_call><|im_end|>

```

That `<tool_call>...</tool_call>` wrapper is Qwen’s convention. And here’s the problem: it’s only Qwen’s convention. Llama wraps calls with its `<|python_tag|>` token. Mistral emits `[TOOL_CALLS]` followed by a JSON array. Gemma, DeepSeek, GLM: each family has its own delimiters, its own payload format, its own quirks. The tool call format is baked into the model at training time, so there’s no way around it: to turn a generation into a structured function call, *someone* has to parse the model-specific format.

Who does the parsing?

It depends on where inference happens:

- Third-party providers: parsing is their problem, not yours. You send `tools=[...]`, you get structured function-call objects back. OpenAI parses its own models' format server-side, and nobody outside even knows what GPT's tool tokens look like.
- Inference servers like vLLM and llama.cpp: they solve it with brute force. Both maintain a collection of per-model tool-call parsers (vLLM's `--tool-call-parser hermes|llama3_json|mistral|...` flag is exactly this), matched to the model and its tokenizer. It works, but it's a growing pile of format-specific code that has to chase every new model release.
- Local in-process inference with `transformers` or `mlx-lm`: nobody parses for you. If we wanted native tool-call support for arbitrary local models, we'd have to reimplement that same pile of per-model parsers ourselves, and keep it in sync forever.

We didn't want to maintain a parser zoo. So we asked: is there one format that *every* instruction-tuned model already speaks fluently, without any model-specific training convention?

Code as the universal tool-calling language

The answer is obvious to any engineer in 2026: code. Every model worth running has seen millions of Python function calls in training. Writing `get_current_temperature(location="Paris, France")` is something a 4B model does reliably; emitting another model family's special tokens is not.

So for the local backend, we sidestep native formats entirely. The JSON Schema tools from `session.update` are converted into Python-style signatures: each schema becomes an `inspect.Signature` via `signature_from_schema`, and `to_code_prompt()` renders it as a readable `def name(...): """docstring"""` block. Those signatures are injected into the system prompt with a Jinja2 template that asks the model to emit exactly one function call inside `<code>` delimiters.

The same weather call, in our format:

```
<code>
get_current_temperature(location="Paris, France", unit="celsius")
</code>

```

That's it. No JSON escaping, no model-specific tokens, just a line of Python any model can write. On the way out, the streaming loop watches the token stream for the `<code>` delimiter: everything before it is speakable text, and the block itself goes to `extract_function_calls_from_text`, which parses the `name(kwargs)` call, validates the arguments against the registered tool schemas, and converts it into the same `ResponseFunctionToolCall` shape the API backends produce.

If you've been following the agents space, you'll recognize the idea: it's the same insight behind `smolagents`' code-agent approach. Models are better at writing code than at filling out JSON forms.

The three backend paths, side by side

Backend	Who parses	
`responses-api`	The provider	Tools passed natively to `client.responses.cre`
`chat-completions`	The provider/server	Streamed `choices[].delta.tool_calls` fragm
`transformers` / `mlx-lm`	Us	JSON Schema → Python signatures → prompt →

Three completely different mechanisms, and the client sees exactly one thing: `response.function_call_arguments.done`.

Closing the loop

Emitting a tool call is only half of an agent turn. The result has to flow back, and here again we follow the OpenAI Realtime protocol exactly:

Two details in this exchange are deliberate:

- Tools run on *your* side. The server never executes anything, whether the tool is a Python function, an API call, or a Reachy Mini turning its head.
- The result doesn't trigger generation. The `function_call_output` lands in the chat context and stays there; the model only speaks again when the client sends `response.create`, so you decide whether the result deserves a spoken follow-up.

This loop is what turns the pipeline from "voice chat" into an agent runtime. On the Reachy Minis, it's how a spoken "look to your left" becomes a motor command and a spoken confirmation, with the same wire protocol whether the LLM is GPT on OpenAI's servers, Qwen on a Cerebras endpoint, or a 4-bit MLX model on a Mac mini in your living room.

7. Interruption and Barge-In

A voice agent you can't interrupt isn't a conversation, it's a voicemail. Barge-in (the user starting to speak while the assistant is still talking) is table stakes for feeling "alive", and it's also where a lot of voice demos quietly fall apart. Not because stopping the audio is hard, but because of everything that happens *after* you stop it.

Here's the problem. When the user interrupts, the pipeline is mid-flight everywhere at once: the LLM is streaming tokens, the TTS is synthesizing audio, and PCM chunks are already sitting in queues waiting to be sent. If you just stop playback, all of that in-flight work keeps arriving. A second later, the assistant blurts out the tail end of the answer you interrupted. Every voice engineer has heard this bug.

Generation-tagged output

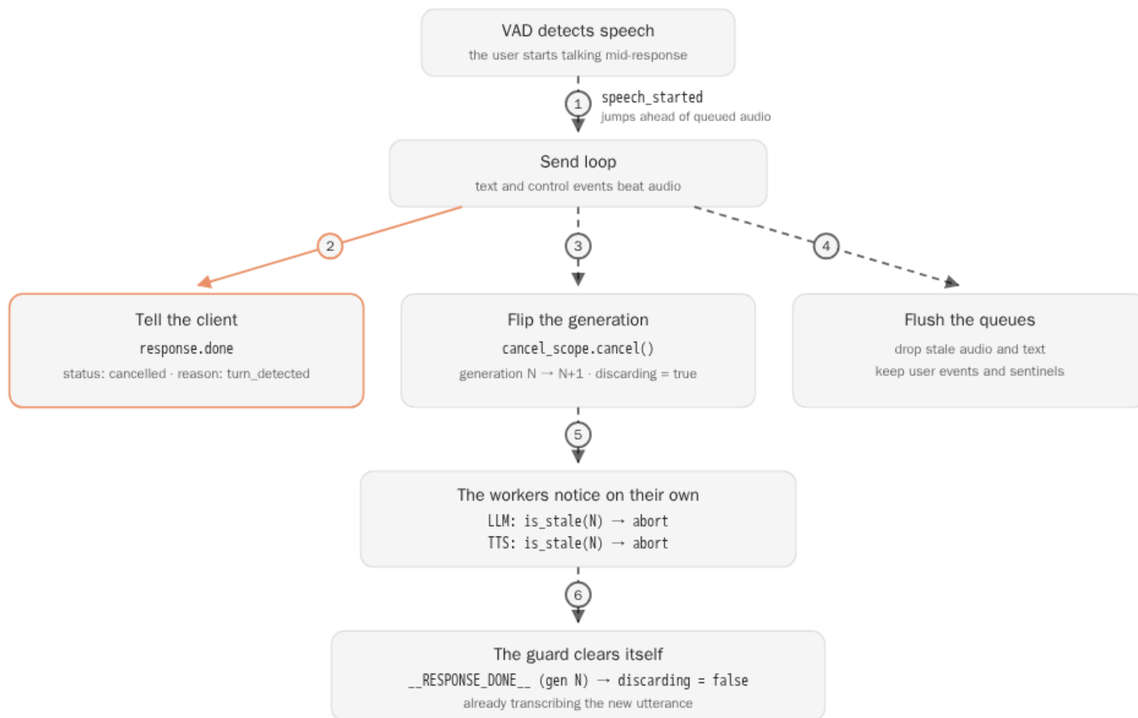
Cancellation lives in a single shared object, `CancelScope`, built around one idea: every response gets a generation number, and every piece of pipeline output is stamped with the generation that produced it.

- When a response starts, the LLM and TTS handlers capture the current generation `N`.
- Every audio chunk and text event they emit carries `cancel_generation = N`.
- Interrupting doesn't "signal" anything. It just increments the counter. Generation `N` is now stale, everywhere, instantly, with no timing games.
- Handlers check `is_stale(gen)` on every streaming token and abort as soon as their generation is superseded.
- The send loop drops any queued item whose generation is stale, so leftovers already sitting in queues never reach the client.

The nice property: output from the *current* generation always passes through. A fresh response can never be swallowed by a lingering discard window from an old one. The generation tag decides what lives and what dies.

What an interruption looks like

Here's the full cascade, from the VAD trigger to a clean pipeline:



A few details took real debugging to get right:

- Interruption is gated. A `SpeechStartedEvent` only triggers a cancel if the session's `turn_detection.interrupt_response` allows it, so you can disable barge-in entirely and the user's speech is still transcribed while the response keeps playing.
- There is one cancellation path, not two. A client-initiated `response.cancel` goes through the exact same machinery as a VAD interruption, just with `reason: "client_cancelled"`.
- A cancel when nothing is playing is a no-op. Without that guard, a spurious VAD trigger (a cough, a door slam) could set a discard guard that no sentinel would ever clear, silently swallowing the next real response.

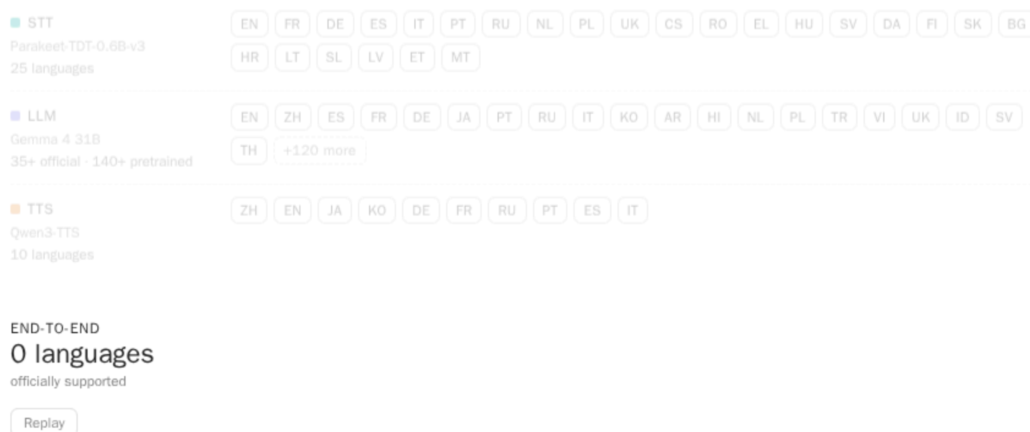
Barge-in is one of those features that's invisible when it works. You talk over the assistant, it stops, it listens, and the conversation just moves on. Nothing about that feels remarkable in the moment, and that's exactly the point.

8. As Multilingual as the Mixture

In the cascade approach, the final product is a multilingual as the intersection of multilinguality in the models you use. For our deployed demo, we chose only multilingual models:

- STT — Parakeet-TDT-0.6B-v3: 25 European languages, with language identification built into the model. The transcript comes back automatically in the spoken language. This may fail, but we are very impressed by the quality of this tiny and fast model. On the downside, because it only supports European languages, the whole pipeline doesn't officialy support Chinese, Japanese, Korean, or many other languages that we love.
- LLM — Gemma 4 31B: 35+ languages out of the box, pretrained on 140+. The LLM is never the bottleneck in the cascade; whatever the other two stages can handle, it can too.
- TTS — Qwen3-TTS: 10 languages, and a very different shape: Chinese, English, Japanese, Korean, German, French, Russian, Portuguese, Spanish, and Italian.

Match the three lists and the officially supported end-to-end set is 7 languages: English, French, German, Spanish, Italian, Portuguese, and Russian. The STT contributes no CJK; the TTS contributes none of the smaller European languages. What survives is the overlap.



For plumbing, we let parakeet transcribe in whatever language it identifies, and we then detect it with lingua-py. The LLM usually replies in the language it's spoken to, but for the Qwen3TTS passing the language it has to produce helps to reduce accent.

Honestly, "officially supported" is a floor, not a ceiling. These models work beyond their "official support": Gemma 4 saw 140+ languages in pretraining, Parakeet transcribes close relatives of its training languages surprisingly well, and Qwen3-TTS will read text in languages it was never

advertised for, with an accent. Catalan, Norwegian, or Swiss German dialects won't show up on any of the three model cards, yet conversations in them mostly just work. We don't want to false advertise language support, but don't let the list of 7 stop you from trying yours.

And finally, the nice thing of the cascade approach is it's flexibility. If you want this to work great for Finish, you can choose models that are great at it. We are trying to cast a wide net here.

9. What This Unlocks

Everything above is plumbing in service of one idea: the realtime voice layer should be flexible for users to customize, and cheap enough to enable real products to be built on top of it.

Because the protocol is OpenAI-compatible and the cascade is fully open, the same server covers a surprisingly wide range of setups:

- A drop-in replacement for OpenAI Realtime. Change the URI and your existing client keeps working. You can even keep GPT-realtime as a fallback while you migrate. That's exactly how we moved the Reachy Mini fleet over, one batch at a time.
- Privacy-sensitive voice agents. Healthcare, legal, anything with kids in the room. With the local setup, audio never leaves your machine. There is no per-minute bill and no third party listening in, because there is no third party.
- On-prem deployments. The whole stack runs in a Docker container on your own GPUs. If your compliance team won't let conversation audio cross a network boundary, that's now a configuration choice instead of a blocker.
- LLM experimentation without touching the voice layer. First-token latency varies wildly between providers serving the same model, and in voice you hear every millisecond of it. Being able to hop from `:cerebras` to `:groq` to a local vLLM server by editing one flag means you can actually feel it.
- Demos that grow into products. The MacBook demo you build with `--local_mac_optimal_settings` speaks the same protocol as our production fleet serving thousands of conversations a day. Going from prototype to deployment is easy.

Where this goes next

We’re serving 15k hours of conversation a month at $0.25/hour, down from $3/hour on GPT-realtime. We think there’s another 10x on the table, to roughly $0.02/hour, by moving the STT and TTS inference to their own endpoints which can be increased/decreased following demand.

The other direction is protocol coverage. We implemented the 18 events a real voice agent needs; the other ~27 are waiting. If your client depends on an event we don’t emit yet, that’s a well-scoped first PR, and the architecture is built for it.

The code is at huggingface/speech-to-speech, Apache 2.0. Try the demo above, point your existing Realtime client at it, or run the whole thing on your laptop tonight. Then come tell us what you built.

Citation

For attribution in academic contexts, please cite this work as

```
Andres Marafioti, Amir Mahla, Thibaud Frere, Leandro Von Werra, Thomas Wolf (2026). "Hugging Voice".
```

BibTeX citation

```
@misc{marafioti2026_hugging_voice,
  title={Hugging Voice},
  author={Andres Marafioti and Amir Mahla and Thibaud Frere and Leandro Von Werra and Thomas Wolf},
  year={2026},

}
```

Made with ❤️ with

research article template