



BUILDING PRODUCTION VOICE AI AGENTS

Real-Time Speech, Telephony, Multi-Agent
Systems and Evaluation

Building Production Voice AI Agents

Real-Time Speech, Telephony, Multi-Agent Systems
and Evaluation

Adrian Morgan Sebuliba

Contents

	Copyright	7
	Start Here	9
	The System, Complete	13
1	The Call That Felt Slow	17
2	Audio Without the Magic	27
3	Your First Browser Voice Agent	35
4	Measure Before Optimizing	43
5	Streaming Speech	49
6	Knowing When the User Is Done	55
7	Interrupt Me	67
8	Cascaded or Realtime	79
9	When the Agent Does Something	89
10	While the Agent Is Working	101
11	Looking Things Up	109
12	What Survives the Call	119
13	How a Phone Call Reaches an Agent	127
14	Your Agent on a Phone Line	137
15	Outbound Calls and Call State	145

16	Provider Independence	155
17	Transfers and Human Handoff	165
18	One Voice, Too Many Powers	173
19	Handoffs Between Agents	179
20	Capability Isolation and Shared Context	185
21	When Multi-Agent Voice Goes Wrong	191
22	What a Good Voice Agent Means	197
23	Simulated Callers and Sandboxes	203
24	Voice Benchmarks and Human Labels	209
25	Red Teams, Invariants and Gates	215
26	The Leaderboard That Refuses	221
27	Observing a Call End to End	227
28	Failure, Recovery and Degradation	233
29	Latency, Cost and Scale	239
30	The Production Audit	245
31	Stop Hard-Coding Agents	251
32	The Voice Agent Studio	257
33	Versions, Deployments, Canaries and Rollback	263
34	Multi-Tenant Voice SaaS	269
35	Take the Studio to Production	283
	Appendix A: Setting Up the Services	299

Glossary of Names 303

Appendix B: Index 309

Copyright

Building Production Voice AI Agents

Copyright © 2026 Adrian Morgan Sebuliba

Published by IAgentic LLC. iagentic.dev

Start Here

A customer says “I was charged twice for my last invoice.” A cascaded voice agent understands every word in well under a second. The caller still waits more than six seconds before it says anything back.

That silence is where this book starts. By the end you will know exactly where seconds like that go, how to measure them instead of guessing, and how to build the rest of a voice agent, telephony, tools, multi-agent handoffs, evaluation, and a production deployment, on the same foundation.

What this book is for

You can read Python, use a terminal, and you have probably built or used a text-based AI agent before. You have not necessarily built anything that has to listen and speak in real time, over a real phone line, to a caller who cannot see a screen and will not wait politely for a spinner.

This book assumes that starting point. It does not assume you know anything about digital audio, the telephone network, or real-time systems beyond using one. Every idea is built from what the chapter before it already established.

What a voice agent actually is

Two designs run through this book. A **cascaded** agent works in steps: it decides the caller has stopped talking, turns their speech into text, sends the text to a language model, and turns the reply back into speech. A **realtime** agent replaces the middle of that pipeline with one model that listens to audio and answers with audio directly. Chapter 8 compares them properly; most chapters before it use whichever one best isolates the thing being measured.

Both designs share the same hard constraint a text-based agent never has: a phone call has no way to show a caller that something is happening. Silence means either “working on it” or “the call has died,” and the caller cannot tell which.

Why a voice agent is not tested like ordinary code

- **Time is the product.** A correct answer that arrives six seconds late still sounds broken. Chapter 1 measures this before anything else gets built.
- **The network fights you.** A telephone call is not a browser tab with worse audio; it is a different, decades-old network that keeps only part of the human vocal range and encodes everything through a codec chosen in the 1960s.
- **Silence is ambiguous.** A pause could be the caller thinking, the caller finished, or the agent still working. Getting that decision wrong either cuts a caller off mid-sentence or leaves them waiting.
- **A mistake can cost real money before anyone notices.** A tool that books a callback and a tool that issues a refund look identical to a language model unless something outside the model stops it from treating them the same way.

Two ideas that run through every chapter

Measure before you optimize. Chapter 4 states this as its own thesis, and the rest of the book follows it: every claim about speed, cost, or correctness in this book comes from a real recorded run, with the number of calls it rests on printed next to it, never from what seemed likely.

The failures stay in. When a run goes wrong, this book shows the wrong number and explains why it was wrong, rather than quietly replacing it with a clean one. A production system fails in the same way; a book that only shows the clean runs would leave you unprepared for the messy ones.

How the book works

Some boxes are for you:

 In plain words

A short, everyday explanation of an idea before it gets used properly.

Most chapters end with the same three sections: **Production pattern**, the durable lessons worth carrying into your own system; **What this chapter did not measure**, stated as plainly as what it did; and a **Remember this** callout, a short list of the chapter's real findings.

Where this book fits

This is the fifth book in the *Production AI Agent Engineering* series. The first four build and evaluate text-based agents and agent products. This book is the first to add a voice: real-time speech, a real telephone network, multi-agent handoffs that have to survive being spoken instead of typed, and the evaluation and production discipline the rest of the series already established, applied to a system that now has to listen and speak in real time.

You can read it without the others. Nothing here assumes you have read a prior book in the series.

What this book leaves out

- **Non-English languages.** Every measurement in this book uses English speech. The endpointing, turn-detection, and codec constraints it measures are not specific to English, but the numbers are.
- **Managed voice-agent platforms.** Vapi, Retell, Bland, and similar platforms let you configure a voice agent through a dashboard and an API, without touching SIP, RTP, or a media server directly. This book teaches the self-hosted path on purpose: understanding what a managed platform hides underneath is what lets you judge whether to trust it. If you want the dashboard-first path instead, start with that platform's own documentation, not this book.
- **Training or fine-tuning a model.** Every model this book uses is called as-is. A tuned model is a different model, and everything measured here would need to be measured again against it.
- **Video or any other modality.** This book is audio only, start to end.

Where a chapter comes near one of these, it says so.

Which chapters need an account, and which don't

Every chapter needs a Gemini API key, free to create in Google AI Studio, with its own free daily quota per model. Beyond that:

- **Chapters 1 through 13** need nothing but Docker, to run the LiveKit media server locally. No telephony account, no cloud account.
- **Chapter 14 onward** adds a Twilio account, free to create, for a real SIP trunk. A real phone number and real call minutes are billed by the carrier once you place one.
- **Chapter 15** adds an Oracle Cloud account, on its Always Free tier, genuinely free at the scale this book uses.
- **Chapter 16** adds a Telnyx account, free to create, to prove nothing in the code is specific to one carrier.
- **Chapter 34** adds an Auth0 tenant, free to create, for real identity and organization membership.

Appendix A lists every service in one place, what each one costs, and which chapter has the full setup. Every recorded run in this book ships in the repository, so you can read and verify every number here without creating a single account.

Reading start to end, or skipping ahead

This book is one continuous build, not a set of stand-alone chapters. Chapter 14’s phone leg assumes Chapter 9’s tools and Chapter 6’s endpointing already exist; Chapter 31’s data-driven agent assumes the whole tool-and-ledger foundation under it. Read start to end for the path this book intends.

If you already know digital audio or basic telephony cold, Chapters 2 and 13 will move quickly for you, and you can skim them without losing the thread of what comes after; each later chapter states plainly what it needs from the ones before it.

Running the code

The code for the whole book is one repository, [voice-agents](https://github.com/IAgentic-LLC/voice-agents). Each chapter is a *tag*, a fixed point in the repository’s history: `ch01-end` is where Chapter 1 finishes, and every later chapter names its own. Git will call this a “detached HEAD” state, which is normal here, since you are looking at a fixed point and not changing it.

```
git clone https://github.com/IAgentic-LLC/voice-agents
cd voice-agents
git checkout ch01-end
uv sync
uv run pytest -q
```

The tests need no key and no account. Every framework version named in this book is the one actually used to produce its numbers, most centrally LiveKit Agents 1.8.2, pinned throughout; provider models are named exactly, since providers rename and retire them often. This book was written and tested against the versions it names in September 2026.

Where you will end up

You will be able to build, measure, and defend a real voice agent: how long a caller actually waits and why, whether a tool call before you meant to run it, whether a keypress arrived as a fact or a guess, whether a handoff between two agents carried the right context, whether a canary release is safe to promote, and whether the whole system survives a real phone call once its favorite failure has been switched off. Each of those has its own chapter, and the Glossary at the back names every term along the way.

The System, Complete

Every chapter in this book adds one working piece to one real system. It is easy to lose the whole shape while reading about one piece of it, so this page exists to hold the whole shape in one place.

The system, complete

Every real component this book built, wired the way it actually connects.

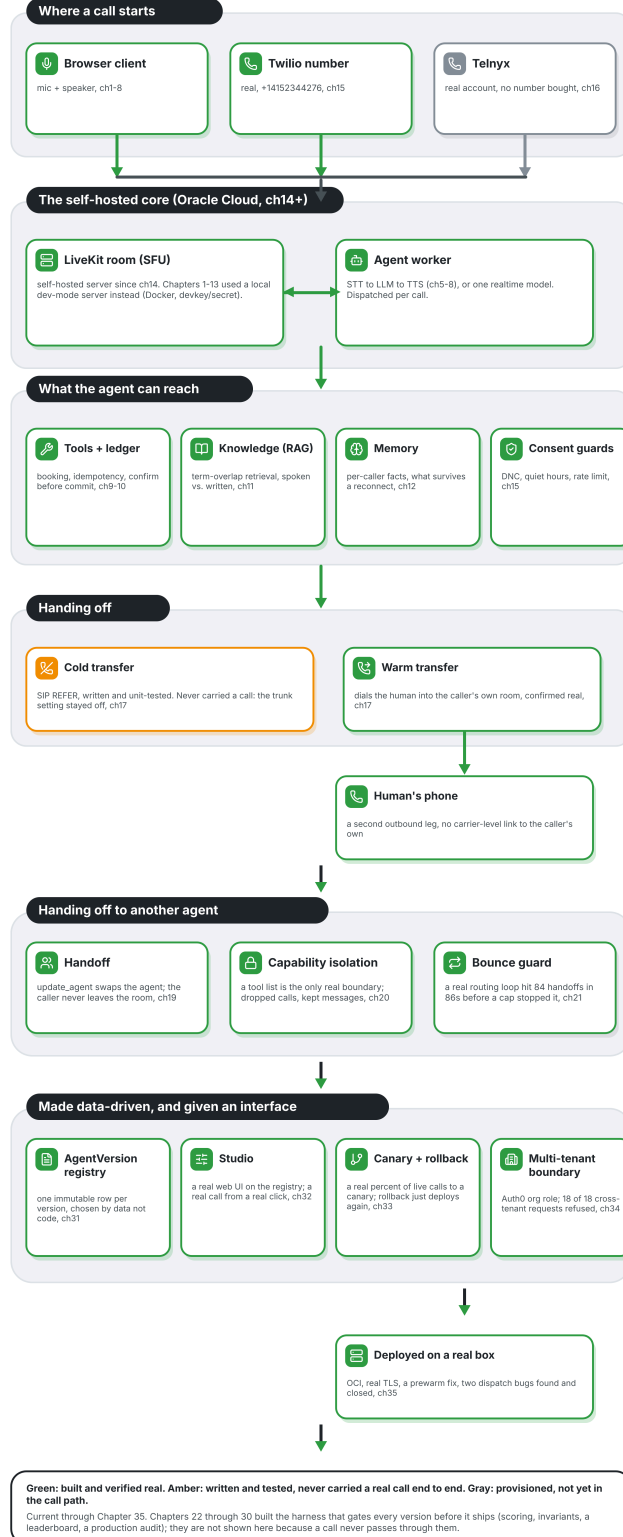


Figure 0.1: Every real component this book built, from the first call to a production deployment, wired the way it actually connects

Figure 0.1 is not a plan. Every green box in it has a chapter number next to it because every green box was built, run, and measured before it earned the color. Amber means the code exists and is tested, but has not yet carried a real call the way the rest of the diagram has. Gray means the account exists and nothing has dialed through it yet.

Two zones do not appear as boxes a call passes through, on purpose. Chapters 22 through 30 built the evaluation harness, scoring, invariants, a leaderboard, a production audit, that gates every version before it ships. It is real, it is load-bearing, and it is not part of the call path itself, so it is not drawn as one.

This page is the finished shape of the system this book built, current through Chapter 35.

1. The Call That Felt Slow

I built two voice agents to answer the same question from a customer who was charged twice. Both answered every call. One answered in about a second and a half. The other left a silence of more than six seconds, long enough to wonder whether the line had dropped.

This chapter measures that silence. It also shows the three wrong numbers I got on the way, because each one teaches something the rest of the book relies on.

1.1 - The problem

Nobody on a phone call waits politely for six seconds. People keep the gaps between turns short. A study of ten languages, from small indigenous communities to major world languages, found that every one of them avoids overlapping talk and keeps the silence between turns to a minimum. The average gap differed from language to language, but only within 250 milliseconds of the overall average, and the authors conclude that people are sensitive enough to those small differences to perceive whole conversational styles as fast or slow.¹

If a quarter of a second is noticeable, six seconds is a broken call. A voice agent is judged against that habit, and a correct answer that arrives late still sounds wrong. So before this book adds tools, telephones or a second agent, it needs one number it can trust: how long the caller waits.

¹Stivers, T., Enfield, N. J., Brown, P., et al. (2009). Universals and cultural variation in turn-taking in conversation. *Proceedings of the National Academy of Sciences*, 106(26), 10587 to 10592.

1.2 - The picture

The smallest voice conversation

Between the caller finishing a question and the first sound of the reply.

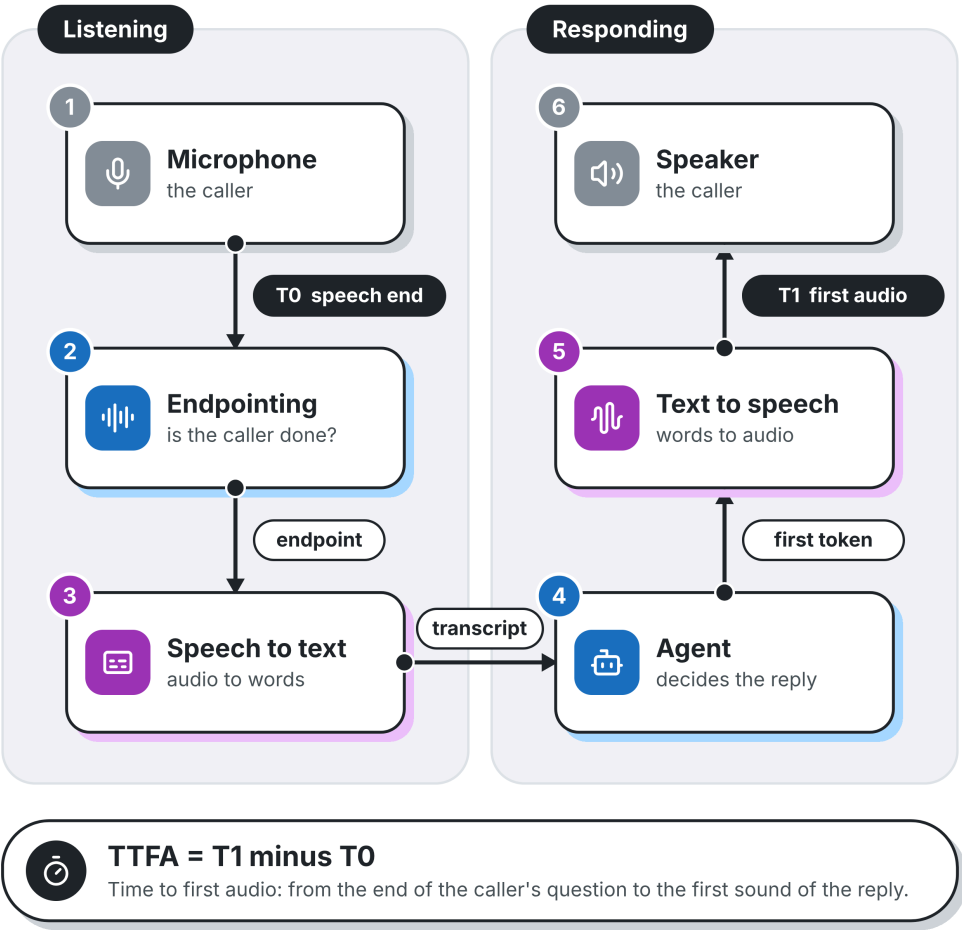


Figure 1.1: The smallest cascaded voice conversation, and the two moments that define the wait

Figure 1.1 shows the parts of a voice agent that the caller never sees. Each one takes time, and the caller only hears the total.

1.3 - The simple explanation

I measure one thing: **time to first audio**, written TTFA. It starts when the caller stops speaking (T0 in the figure) and ends when the agent's first sound reaches the caller (T1).

💡 In plain words

TTFA is the silence after you finish a sentence and before the other side starts to answer. On a good phone call it is short enough that you never notice it.

Two designs sit behind this chapter. The figure shows a **cascaded** agent, which works in steps. It first decides that the caller has finished speaking, a step called *endpointing*. Then it turns the speech into text, sends the text to a language model, and turns the model's reply back into speech. A **realtime** agent replaces the middle of the figure with one model that listens to audio and answers with audio. Chapter 8 compares them properly. Here I only need both to exist, so I can measure them the same way.

1.4 - Under the hood

A human caller is hard to repeat, so the code uses a synthetic one. It connects through LiveKit, the open-source media server this book uses. LiveKit carries audio between participants in a *room*, the way a conference call does. `caller.py` creates a fresh room for each call, joins it, asks an agent to join too, and plays a recorded question. The recording is 6.8 seconds long:

Hi, I was charged twice for my last invoice. Can you tell me how a refund works?

The words end at 6.0 seconds. The last 0.8 seconds of the file is quiet room tone, and T0 is the end of the words, not the end of the file. The caller marks T0 when the last loud frame of the question goes out.

Audio travels as *frames*, short slices of sound. Here each frame is 10 milliseconds, so one second of speech is 100 frames. The caller sends the question frame by frame, then keeps sending silent frames while it listens.

It counts an agent frame as sound once its loudness passes a fixed threshold, so near-silent frames do not count as an answer. Loudness here is the root mean square of the samples: square each sample, average them, and take the square root, which measures how far the wave swings either way. A *sample* is one measurement of the sound wave, and the agent's audio arrives as 24,000 samples a second. Each sample is a 16-bit number, so it runs from -32,768 to 32,767. The threshold is 300, which is my choice, and a different threshold would move every result slightly.

Each call is one line in a *run*, a folder of records. The agent writes its own timings to the same folder, so I can see where the time went and not only how much there was.

1.5 - Build it

The code is in the `voice-agents` repository at the tag `ch01-end`, a fixed point in its history. It needs Python 3.12 or newer and `uv`, a tool that installs and runs Python packages. Install `uv` once. On macOS or Linux run `curl -LsSf https://astral.sh/uv/install.sh | sh`, and on Windows run `winget install --id=astral-sh.uv -e`. Then:

```
git clone https://github.com/IAgentic-LLC/voice-agents
cd voice-agents
git checkout ch01-end
uv sync
uv run pytest -q
```

Git will say you are in a “detached HEAD” state. That is normal here, because you are looking at a fixed point and not changing it. The code uses LiveKit Agents 1.8.2.

The tests need no key and no account, because the runs I recorded ship with the code. This command prints the realtime run:

```
uv run report.py runs/ch01-realtime
```

```
== runs/ch01-realtime
calls 10, agent joined 10
answered 10 of 10 (95% interval 72.2% to 100.0%)
time to first audio, median 1.492 s (95% interval 1.422 to 1.547 s)
  fastest 1.406 s, slowest 1.594 s
```

The realtime agent uses `gemin-3.8-live`. It answered all ten calls, and the typical caller waited about a second and a half. Two parts of that output need explaining.

The *median* is the middle result: half the calls were faster and half slower. I use it instead of the average because one slow call can drag an average a long way.

The *95% interval* is the range of true medians that these ten calls are consistent with. It describes how sure the evidence lets me be about this agent under these conditions. It is not a forecast of the next ten calls. The next section shows how it is made.

To place your own calls you need Docker, which runs the LiveKit server (Docker Desktop on Windows and macOS), and a Gemini API key from Google AI Studio. Copy `.env.example` to `.env` and put your key after `GEMINI_API_KEY=` on its one line. Never commit that file. Then start the server:

```
docker run -d --name livekit -p 7880:7880 -p 7881:7881 \
  -p 7882:7882/udp livekit/livekit-server:latest \
  --dev --bind 0.0.0.0 --node-ip 127.0.0.1
```

The backslash at the end of a line continues a command in bash, which is what Git Bash on Windows uses. In PowerShell, type the command on one line without the backslashes. Next, start an agent in one terminal and leave it running:

```
uv run realtime_agent.py start
```

The agent runs as a *worker*: a process that waits for LiveKit to hand it a call. It is ready when its log shows `registered worker`. In a second terminal, place ten calls and read the report:

```
uv run caller.py --agent realtime --calls 10 --run runs/mine-realtime
uv run report.py runs/mine-realtime
```

The cascaded agent works the same way. Start `uv run cascaded_agent.py start` instead, and call it with `--agent cascaded`. The later sections name the one setting each of its runs changed. Stop a worker with `Ctrl+C`, and the server with `docker stop livekit`.

Your numbers will differ from mine. The model is not deterministic, meaning the same input can produce a different output, and your machine and network are not mine. Expect the same shape, not the same digits.

1.6 - How much evidence is ten calls?

Ten out of ten answered sounds like 100%. With ten calls, the true answer rate could still plausibly be as low as 72.2%, which is the lower end of the interval in the output. *Evaluating AI Agents*, the fourth book in this series, is built on the same rule: a score is reported with the range the evidence supports, never alone.

i Under the hood: two kinds of interval

For the answer rate I use the Wilson score interval, which stays sensible when every call succeeds. For the median wait I use a *bootstrap*. It builds a new set of ten by drawing from my ten at random, where the same call can be drawn more than once, and takes that set's median. It does this 10,000 times. The middle 95% of those medians is the interval. The random draws start from a fixed seed, a starting number for the random generator, so the report prints the same interval every time. Both functions are in `voicelab/stats.py`, and `tests/test_stats.py` checks them.

1.7 - What breaks: three wrong numbers

Three numbers I produced while building this chapter were wrong. The way each one was wrong is the most useful part of this chapter.

1.7.1 - The clock that started at the end of the file

My caller first started its clock when the recording finished playing. That sounds like the end of the question. It is not: the file runs 0.775 seconds past the last word, into quiet room tone. Every wait it reported was about 0.78 seconds short, and the realtime agent looked like it answered in 0.727 seconds. Each record in `runs/ch01-realtime/trials.jsonl` still holds that measurement as `ttfa_s`.

Nothing about the number looked wrong. It was fast, stable across ten calls, and had a tight interval. I found the problem only when I measured the same agent from a web browser, in Chapter 3. The browser's measurement started from the last loud moment of speech, and the two disagreed by about 0.8 seconds. Measured the same way, they agree.

The fix is the definition in the section above: `T0` is the moment the last loud frame goes out. Each record also holds `question_s`, how long the whole file took to play, 6.765 to 6.781 seconds here. From that, `report.py` converts the older records exactly, and every number in this chapter counts from the end of the words.

This is the first rule of the book: before trusting a measurement, test the instrument that makes it. Two instruments that disagree are the fastest way to find out that one of them is wrong.

1.7.2 - The question cut in two

The cascaded agent uses `gemini-3.5-flash-lite` both to transcribe and to reply, and a text-to-speech (TTS) model for its voice. Transcription here is *batch*: the agent waits until a stretch of speech is over, then sends the whole stretch at once. This run used the agent's default settings. It came back faster than the next one, an early sign something was off:

```
uv run report.py runs/ch01-cascaded-split
```

```

== runs/ch01-cascaded-split
calls 10, agent joined 9
answered 9 of 9 (95% interval 70.1% to 100.0%)
time to first audio, median 6.047 s (95% interval 5.922 to 6.484 s)
  fastest 5.734 s, slowest 6.750 s
  transcription (batch)    1.243 s median
  model first token        0.641 s median
  speech first byte        2.936 s median
model calls: 18 in 9 calls
heard 9 times:
  Hi, I was charged twice for my last invoice.
heard 9 times:
  Can you tell me how a refund works?

```

Three lines in that report are stages of the pipeline. *Transcription* is the time to turn a stretch of speech into text. *Model first token* is the wait for the first piece of the model's reply. A *token* is the unit a language model reads and writes: a short word, or a piece of a longer one. *Speech first byte* is the wait from sending the reply text to the voice model until the first piece of audio, the first byte, comes back. The stages overlap in time, because the agent starts speaking the first sentence while the model is still writing the rest, so they do not add up to the total.

The last lines show what the agent heard. It never heard the question as one question. It heard two, and it called the model twice per call.

The cause is a pause. The recording has 0.90 seconds of silence between its two sentences. The agent's endpointing starts with a *voice activity detector*, a small model that marks where speech starts and stops. This one is called Silero, and the agent loads it as a plugin. By default it ends a stretch of speech after 0.55 seconds of silence. So it cut the question at the pause, and the agent transcribed and answered each half on its own.

In plain words

Imagine a waiter who walks off to the kitchen the moment you pause to breathe. "I would like the soup..." and he is gone, before you could say "...without onions".

To see what happened inside one call, `timeline.py` lists each event in order. Without `--call` it prints every call in the run; with it, the one named. Times are seconds from the moment the caller stopped speaking, so a negative time happened while the caller was still talking. These runs did not record that moment, so the script estimates it from when the call's record was written. The estimate can run a few tenths of a second late, so the times shown may be a few tenths too early. The order of events is exact. The caller now records the moment, so later runs will not need the estimate.

```
uv run timeline.py runs/ch01-cascaded-split --call 2
```

```

call 2: answered, 6.047 s
-0.77 s heard Hi, I was charged twice for my last invoice.
-0.00 s model 29 prompt tokens
+1.99 s heard Can you tell me how a refund works?
+2.01 s speech 34 chars, 0.0 s audio, cancelled
+3.21 s model 38 prompt tokens
+6.04 s speech 34 chars, 2.5 s audio, finished
+12.89 s speech 165 chars, 9.5 s audio, finished

```

The *prompt* is everything sent to the model for one reply. The first model call had 29 prompt tokens: the agent's instructions plus the first sentence alone. The agent had an answer to it

around the moment the caller finished speaking. When the second sentence arrived, the agent cancelled the speech for that answer before any of it played, and asked the model again with the instructions and both sentences, 38 tokens. The caller heard only the second answer. The same thing happened in all nine calls: every answer to half the question was written and thrown away, with no audio played.

So the caller got a sensible answer, and the agent paid for a model call nobody heard. The run looks faster than the correct one below only because its turn ended sooner. It waited 0.55 seconds of silence after the question instead of 1.2.

Setting the detector to wait 1.2 seconds, longer than the pause, keeps the question whole. This run started the agent with `VAD_SILENCE=1.2 uv run cascaded_agent.py start`. In PowerShell, set the variable first with `$env:VAD_SILENCE = "1.2"`, then start the agent.

```
uv run report.py runs/ch01-cascaded-whole
```

```
== runs/ch01-cascaded-whole
calls 10, agent joined 10
answered 10 of 10 (95% interval 72.2% to 100.0%)
time to first audio, median 6.624 s (95% interval 6.367 to 6.703 s)
  fastest 6.156 s, slowest 7.140 s
  transcription (batch) 1.281 s median
  model first token      0.671 s median
  speech first byte      2.897 s median
model calls: 10 in 10 calls
heard 10 times:
  Hi, I was charged twice for my last invoice. Can you tell me how a
  refund works?
```

One model call per call, and the whole question every time. It is slower by 0.58 seconds at the median, close to the 0.65 seconds of extra silence the detector now waits. That wait is added to every turn, and a speaker with a longer pause would still be cut off. Chapters 5 and 6 replace this blunt fix with streaming transcription and a turn detector that looks at what was said, not only at silence.

In the split run, one call never got an agent: a worker process did not start in time, and the caller gave up after 40 seconds. The report counts answers only among the calls the agent joined, and it prints both counts.

1.7.3 - The fast run that answered the wrong question

The largest stage in both cascaded runs is speech first byte, about 2.9 seconds.

That voice model is `gemini-2.5-flash-preview-tts`, and I did not start with it. I started with `gemini-3.1-flash-tts-preview`. The first cascaded run I recorded used it, with the default detector, started with `TTS_MODEL=gemini-3.1-flash-tts-preview uv run cascaded_agent.py start`. From its fourth call on, speech requests began to fail:

```
Quota exceeded for metric: generate_requests_per_model_per_day,
limit: 100, model: gemini-3.1-flash-tts
```

The key I used allows 100 speech requests a day for each voice model. A cascaded call makes about two, because the agent sends each sentence of its reply as its own request. After a day of experiments I had used them up. Calls 4, 6, 8, 9 and 10 got silence. Calls 5 and 7 still got through.

That run is in the repository as `runs/ch01-tts-daily-limit`, and it holds the third wrong number. Its report shows a median wait of 2.125 seconds, which looks far better than either run above. It is wrong twice over. The median covers only the five calls that got any answer, while the other five got silence. And the run used the default 0.55-second detector, so it heard the question

in two halves. With this faster voice, the speech first byte is 0.853 seconds, and the answer to half the question started playing before the agent had caught up with the second half:

```
uv run timeline.py runs/ch01-tts-daily-limit --call 3
```

```
call 3: answered, 2.125 s
-0.80 s heard Hi, I was charged twice for my last invoice.
+0.03 s model 29 prompt tokens
+1.86 s heard Can you tell me how a refund works?
+2.20 s speech 34 chars, 1.9 s audio, cancelled
+3.55 s model 41 prompt tokens
+6.55 s speech 99 chars, 5.6 s audio, finished
+8.70 s speech 66 chars, 4.4 s audio, finished
```

The voice model produced 1.9 seconds of audio for that first answer, and the caller’s first sound, at 2.125 seconds, came from it. Then it was cut off. The records show how much audio was produced, not how much of it played before the cut. In all five answered calls, between 1.5 and 2.9 seconds of audio was produced for the answer to “I was charged twice” before the agent answered again. In four calls it was cut off. In call 5 its first sentence played to the end and only its second sentence was cancelled. The second model call here had 41 prompt tokens, not 38, most likely because the spoken part of the first answer stayed in the conversation. The 2.125 seconds measured how fast the agent began answering the wrong question.

One split question, two voices

The same fault, heard or hidden. Order of events, not to scale.

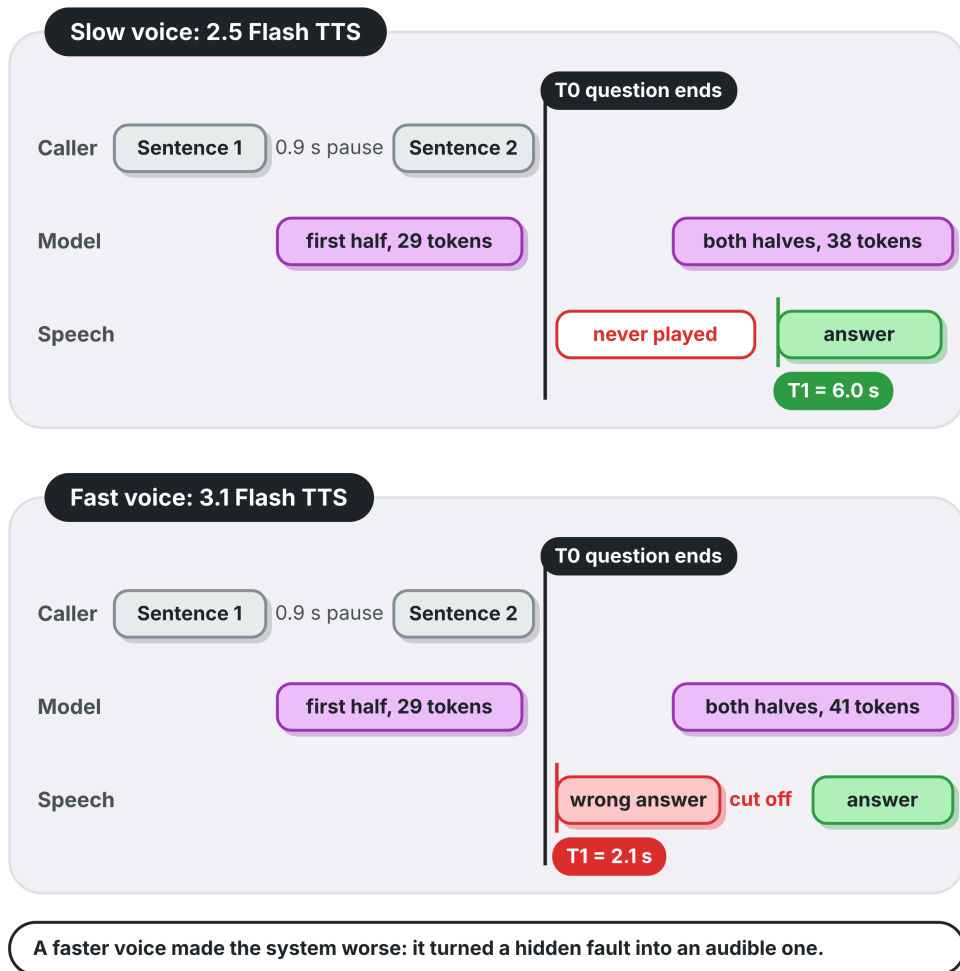


Figure 1.2: The same split question with a slow voice and a fast voice

The same fault, a pause that splits the question, stayed hidden with a slow voice and became audible with a fast one. Figure 1.2 puts the two calls side by side. A faster part made the system worse. That is why the report prints what the agent heard and how many calls every number covers, and why `timeline.py` exists.

I switched to the 2.5 voice model, which has its own daily limit and half the price per audio token. It is also more than three times slower to start speaking, which cost about two seconds on every reply. Check the limits on your own key in Google AI Studio before a long session. Failures from a limit look like an agent that ignores the caller, not like an error message.

1.8 - What this chapter did not measure

- **One machine.** Everything ran on one Windows laptop against a local server. A real network, a phone line or a cloud server adds delay that this chapter does not include. Under load, the worker marks itself too busy and refuses calls, so these numbers assume a mostly idle machine.
- **One voice, one question.** The caller is synthetic speech, played perfectly. Real callers mumble, talk over the agent and pause in odd places.
- **Only the start of the answer.** The caller listens for 12 seconds after the question. A slow agent's later sentences can fall outside that window, so the recorded answer length undercounts slow agents. This chapter does not use it.
- **Not a verdict on designs.** The realtime agent was about four and a half times faster to first audio here. That is one question on one day, with one choice of models for the cascaded agent. Chapter 8 compares the designs on more than speed.

Models are paid by the token. At the time of writing, Google lists [gemini-3.8-live](#) at \$3.00 per million tokens of audio sent to the model, \$0.75 per million tokens of text sent, and \$12.00 per million tokens of audio it returns.² At those prices a realtime call here cost about \$0.0036 at the median, from the token counts the agent logged.

i Remember this

- Time to first audio is the silence the caller hears. Measure it from the end of the caller's speech to the first audible sound of the answer.
- Test the measuring tool first. My first caller started its clock at the end of the file, not the end of the words, and every wait read 0.78 seconds short.
- Read what the agent heard, not only how fast it was. A pause split the question, and a faster voice turned that hidden fault into an answer to the wrong question.
- A number is only as good as the calls it covers. Print how many calls each number rests on.
- Report every number with its interval, the model names and the conditions. Ten calls is a start, not a conclusion.

²Google, Gemini API pricing, ai.google.dev/gemini-api/docs/pricing.

2. Audio Without the Magic

In Chapter 1 I measured how long a caller waits. This chapter looks at what actually travels down the line while they wait. It is a stream of numbers. The numbers carry no meaning without a label saying how fast to play them, and one wrong label made a speech model invent the end of a question.

2.1 - The problem

A voice agent passes audio between parts that expect different formats. Google’s Live API, the realtime model from Chapter 1, takes in audio at 16,000 samples a second and sends its replies at 24,000.³ A telephone line carries 8,000 samples a second.⁴ Every hand-over between two of these parts is a place where the audio must be converted, and where the label saying what the audio is can be wrong.

These mistakes rarely crash anything. The audio still plays, and the agent still answers. That is what makes them dangerous.

³Google, Live API capabilities guide, ai.google.dev/gemini-api/docs/live-guide: “Audio data in the Live API is always raw, little-endian, 16-bit PCM. Audio output always uses a sample rate of 24kHz. Input audio is natively 16kHz.”

⁴ITU-T Recommendation G.711, *Pulse code modulation (PCM) of voice frequencies*: “The nominal value recommended for the sampling rate is 8000 samples per second.” [itu.int/rec/T-REC-G.711](https://www.itu.int/rec/T-REC-G.711).

samples a second. Keep the two apart: a tone’s frequency is how fast the sound vibrates, and the sample rate is how often we measure it.

💡 In plain words

A sample rate is like the frame rate of a film. A film is still pictures shown fast enough to look like motion. Digital audio is measurements taken fast enough to sound like a smooth wave. Show the pictures at the wrong speed and everyone moves too fast or too slow.

Each sample in this book is a 16-bit whole number, from $-32,768$ to $32,767$. A list of such numbers, one after another, is called **PCM**, for pulse code modulation. It is the plain, uncompressed form of digital audio, and it is what the Live API takes and gives. As in Chapter 1, the samples travel in **frames** of 10 milliseconds, so a 16 kHz frame holds 160 samples.

A WAV file is PCM with a small header in front. The header records the sample rate, the number of channels and the bits per sample. Microsoft’s description of the format says the sample rate field “specifies the sample frequency at which each channel should be played or recorded”.⁵ The samples themselves do not know their rate. Only the header says it.

How fast is fast enough? The engineer C. E. Shannon put the rule this way in 1949:⁶

If a function $f(t)$ contains no frequencies higher than W cps, it is completely determined by giving its ordinates at a series of points spaced $1/2W$ seconds apart.

In that sentence “cps” means cycles per second, the old name for hertz, “ordinates” means the measured values, and the spacing is $1/(2W)$ seconds, so W cycles a second need $2W$ samples a second. In today’s words, a signal can hold tones up to half its sample rate. That limit is called the *Nyquist frequency*. At 16 kHz it is 8,000 Hz, and at a telephone’s 8 kHz it is 4,000 Hz. Telephone standards describe the band a phone keeps as 300 to 3,400 Hz.⁷ A tone above the limit does not simply disappear. It folds back below the limit and shows up as a different tone, which is called **aliasing**. A 5,000 Hz tone sampled at 8 kHz sits 1,000 Hz above the 4,000 Hz limit, so it folds to 1,000 Hz below it: 3,000 Hz.

To change a sample rate properly, a **resampler** first removes the frequencies the new rate cannot hold, then keeps fewer samples. Removing frequencies is called filtering. Skipping the filter is the classic mistake.

2.4 - Build it

The code is at the tag `ch02-end` in the same repository. From your `voice-agents` folder:

```
git fetch --tags
git checkout ch02-end
uv sync
uv run pytest -q
uv run audio_lab.py make
uv run audio_lab.py inspect
```

⁵Microsoft, WAVEFORMATEX structure, learn.microsoft.com/en-us/windows/win32/api/mmreg/ns-mmreg-waveformatex.

⁶Shannon, C. E. (1949). Communication in the presence of noise. *Proceedings of the IRE*, 37(1), 10 to 21. Theorem 1. For the folding rule, see J. O. Smith, *Mathematics of the Discrete Fourier Transform*, “Aliasing of sampled signals”, ccrma.stanford.edu/~jos/st.

⁷ITU-T Recommendation P.310, *Transmission characteristics for telephone-band (300-3400 Hz) digital telephones*, [itu.int/rec/T-REC-P.310](https://www.itu.int/rec/T-REC-P.310).

`uv run pytest -q` checks every number in this chapter against the code. `make` writes eight files to `audio/ch02/`, built from the Chapter 1 question and from a test tone. It needs no key and no server. Open them in any audio player and listen before you read the numbers. `inspect` prints what each file contains:

file	rate	seconds	loudness	peak Hz	>3.4k
<code>refund_question.wav</code>	16000	6.78	3164	87	2.5%
<code>question-8k-mulaw.wav</code>	8000	6.78	3126	87	0.2%
<code>question-8k-naive.wav</code>	8000	6.78	3164	87	1.4%
<code>question-8k-resampled.wav</code>	8000	6.78	3127	87	0.2%
<code>question-labelled-24k.wav</code>	24000	4.52	3164	130	3.6%
<code>question-labelled-8k.wav</code>	8000	13.55	3164	43	0.1%
<code>tone-5000hz.wav</code>	16000	2.00	5657	5000	100.0%
<code>tone-8k-naive.wav</code>	8000	2.00	5656	3000	0.0%
<code>tone-8k-resampled.wav</code>	8000	2.00	16	-	19.2%

Rate is the sample rate in the header. *Loudness* is the root mean square from Chapter 1. *Peak Hz* is the loudest frequency, found with a Fourier transform, the standard method that splits a signal into the tones it contains. The last column is the share of the sound’s *energy* above 3,400 Hz, the top of the telephone band. Energy here is the sum of the squared sample values, a measure of how much sound there is, and the Fourier transform tells how it divides among frequencies. The code is in `voicelab/audio.py`, and each function there is a few lines long. The table has nine rows because it lists the original question first. Durations are rounded: the original is 6.776 seconds, so half speed is 13.55 seconds, not 13.56.

The original question’s loudest frequency is 87 Hz, the base pitch of the synthetic voice. Only 2.5% of its energy is above 3,400 Hz. That small share still matters to your ear, because it carries the hiss of sounds like “s” and “f”.

2.5 - What breaks

2.5.1 - The same samples with the wrong label

`question-labelled-24k.wav` and `question-labelled-8k.wav` contain exactly the same numbers as the original. Only the header changed. Labelled 24 kHz, the question plays in 4.52 seconds instead of 6.78, and the voice’s 87 Hz becomes 130 Hz: one and a half times faster and higher. Labelled 8 kHz, it takes 13.55 seconds and drops to 43 Hz: half speed, a deep drawl. Every frequency moves by the same ratio, which is also why the share above 3,400 Hz rises to 3.6% in the fast version and falls to 0.1% in the slow one.

Nothing checks the label against the audio. A player believes the header, and so does a model.

2.5.2 - Too few samples, and no filter

The simplest way to turn 16 kHz into 8 kHz is to keep every other sample. `naive_downsample` does exactly that, and nothing else:

```
def naive_downsample(pcm, factor):
    return pcm[::factor].copy()
```

On the test tone, the result is the picture at the start of this chapter. `tone-5000hz.wav` is a 5,000 Hz tone. After keeping every other sample, `tone-8k-naive.wav` holds a 3,000 Hz tone at the same loudness, 5,656 against 5,657. It is a clean, loud note that the original never contained. Listen to the two files one after the other.

`tone-8k-resampled.wav` went through `resample`, which uses LiveKit’s resampler. LiveKit’s source code says it resamples “using the Sox resampling library”, a long-established open-source

resampler.⁸ It filtered out the 5,000 Hz tone before dropping samples, because 8 kHz cannot hold it. What is left has a loudness of 16 on a scale that runs to 32,767, which is silence. Silence is the right answer here, and the table prints no peak for it. The 19.2% in its last column is a share of almost nothing: a faint remainder near 3,840 Hz, just under the 4,000 Hz limit, where no filter can cut perfectly.

Speech has much less energy up there than a test tone, so the damage is smaller, and it can be traced exactly. The naive 8 kHz question has 1.4% of its energy between 3,400 and 4,000 Hz, against 0.2% for the resampled one. In the original, only 0.26% of the energy is in that band, and 1.11% is just above it, between 4,000 and 4,600 Hz. At 8 kHz that second band folds onto the first, and 0.26% plus 1.11% is 1.37%. So most of the naive file's top band is hiss that was above 4,000 Hz in the original, folded down to where it never was.

2.5.3 - What the model heard

Your ear hears the difference between these files at once. The question for a voice agent is whether a speech model does. `transcribe_variants.py` sends each version five times to `gemini-3.5-flash-lite`, the model the Chapter 1 cascaded agent uses to transcribe, and scores each transcript with a word error rate.

In plain words

The **word error rate** counts the fewest word changes (swaps, deletions and insertions) that turn the transcript into what was really said, divided by the number of words really said. Zero is perfect. A rate of 0.29 means about three words in ten are wrong.

The recorded run is in the repository, so the report needs no key. To record your own, with the key from Chapter 1, run `uv run transcribe_variants.py --repeats 5 --run runs/mine-transcripts`. This is the report for the run I recorded:

```
uv run transcribe_variants.py --report runs/ch02-transcripts
```

```
version                tries  perfect  mean WER
refund_question.wav    5      5      0.000
question-8k-resampled.wav 5      5      0.000
question-8k-naive.wav   5      5      0.000
question-8k-mulaw.wav   5      5      0.000
question-labelled-24k.wav 5      5      0.000
question-labelled-8k.wav 5      0      0.259
perfect overall: 66.4% to 92.7% (95%)
worst (question-labelled-8k.wav, wer 0.29):
  I was charged twice for my last invoice. Can you tell me how we can
  resolve this?
```

The model shrugged off everything my ear objected to. Telephone rate, naive or resampled, was perfect. So was the telephone codec. A *codec* is a method for packing audio into fewer bits and unpacking it again. G.711 packs each sample into 8 bits with a curve called μ -law, which keeps quiet sounds more precisely than loud ones, because ears notice errors in quiet sounds more. `question-8k-mulaw.wav` went through that packing and unpacking. Even the question played 1.5 times too fast came through word for word.

The half-speed question failed every time, and never with gibberish. All five transcripts were fluent English. Four of them dropped “Hi”, and all five rewrote the end: “how we can resolve this” three times, “how we fund what” once, and “how we can work” once. The caller asked how a

⁸LiveKit, `livekit-rtc/livekit/rtc/audio_resampler.py`, github.com/livekit/python-sdks.

refund works. An agent reading this transcript would answer a different question, confidently, and nothing in the transcript would look wrong.

That is the failure to design against. A garbled transcript is easy to catch. A fluent one that says something the caller never said is not.

The “perfect overall” line is the 95% interval from Chapter 1, for the share of perfect transcripts across all 30 tries, and it is wide because one version failed every time. Five tries per version is enough to show a version that always fails. It is not enough to promise that the others never do.

2.6 - Production pattern

The fixes are simple, and each one has to be applied at every boundary where audio changes hands.

- **Say the rate wherever audio crosses a boundary.** The Live API asks for the rate in the audio’s type, for example `audio/pcm;rate=16000`, and resamples on its side if needed.⁹ That only works if the stated rate is the true one.
- **Resample with a real resampler.** Use `rtc.AudioResampler` or another filtered resampler. Never keep every other sample.
- **Check the rate where audio arrives.** LiveKit frames carry their own `sample_rate`. Compare it with what the next stage expects, and fail loudly on a mismatch rather than playing the audio anyway:

```
def expect_rate(frame, rate):
    if frame.sample_rate != rate:
        raise ValueError(
            f"expected {rate} Hz audio, got {frame.sample_rate} Hz"
        )
```

- **Test with a known tone.** Push a tone of known pitch through the pipeline and check the pitch that comes out, as `tests/test_audio.py` does. A wrong label or a missing filter moves the pitch, and the test catches it without any listening.
- **Keep a known question in the tests.** A question whose words you know, transcribed with a word error rate of zero, is a cheap check that audio reaches the model in the right shape.

2.7 - What this chapter did not measure

- **One voice.** The question is synthetic speech. Real voices, background noise and cheap microphones put more energy in more places.
- **One model, on one day.** Today’s model handled telephone-quality audio perfectly. An older transcriber, or a different one, may not, so check the model you use.
- **A stand-in for the telephone codec.** `mulaw_roundtrip` uses the continuous μ -law formula. G.711 itself uses a close table-based version of it.
- **No compressed codecs.** WebRTC, which browsers use for calls, requires both G.711 and the Opus codec, and Opus can run at 8, 12, 16, 24 or 48 kHz.¹⁰ This chapter did not test Opus.

⁹Google, Live API capabilities guide, ai.google.dev/gemini-api/docs/live-guide: “Audio data in the Live API is always raw, little-endian, 16-bit PCM. Audio output always uses a sample rate of 24kHz. Input audio is natively 16kHz.”

¹⁰IETF RFC 7874, *WebRTC Audio Codec and Processing Requirements*, and RFC 6716, *Definition of the Opus Audio Codec*, rfc-editor.org.

i Remember this

- Digital audio is a list of numbers plus a label saying how fast to play them. The numbers do not know their own rate.
- A signal holds tones up to half its sample rate. Anything higher folds back as a tone that was never there.
- Resample with a filter. Keeping every other sample invents sounds.
- The worst audio fault is not noise. It is a wrong label that turns speech into a fluent, wrong transcript.
- Test the pipeline with a tone of known pitch and a question of known words.

