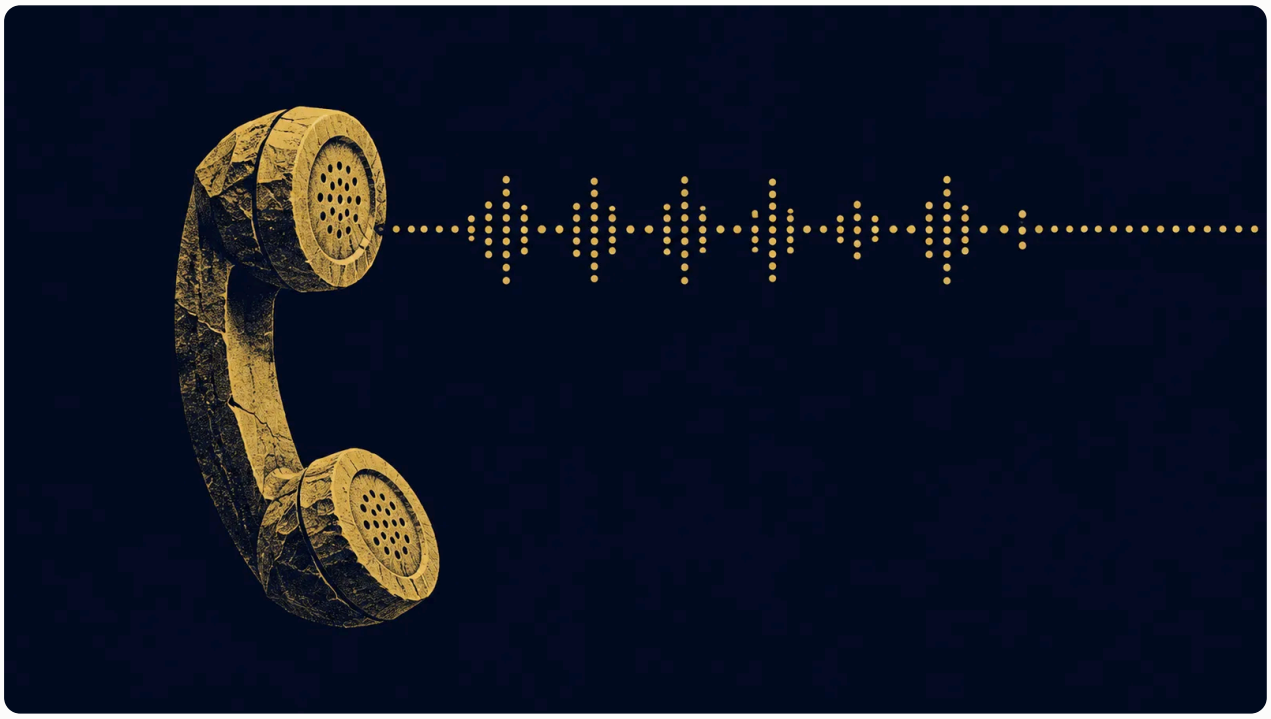
AMEN AMOUZOU · AUGUST 7, 2026

Giving AI agents a real phone line

A browser voice demo proves your model can talk. It doesn't prove your stack can survive a phone network — so I measured mine: frame-timing distributions at 50 concurrent calls, plus a live-trunk check.



Between a WebRTC tab and a real phone call sit thirty years of SIP signaling, RTP media, NAT traversal, and codec negotiation — and the interface that actually reaches everyone (your grandmother, a contractor on a job site, a customer rescheduling a delivery) lives on the far side of all of it.

Over the last six months, the stack described in this article has carried **more than half a million production phone calls** — real customers, real trunks, real hold music. But “it

works in my production" is a claim, not evidence. So this time I measured the thing that decides whether a phone call can feed an AI pipeline: **does the audio arrive steadily, and does it stay steady under load?**

Why frame timing is the question

Strip away the telephony folklore and an AI agent needs three things from a phone call: to **answer or dial**, to **hear** (a stream of decoded audio frames), and to **speak** (a stream of frames going back). That's the entire surface of xphone-go:

```go
call, _ := phone.Dial(ctx, "+14155550142")
for frame := range call.PCMReader() {
    stt.Write(frame)    // your STT, your LLM, your voice
}
```

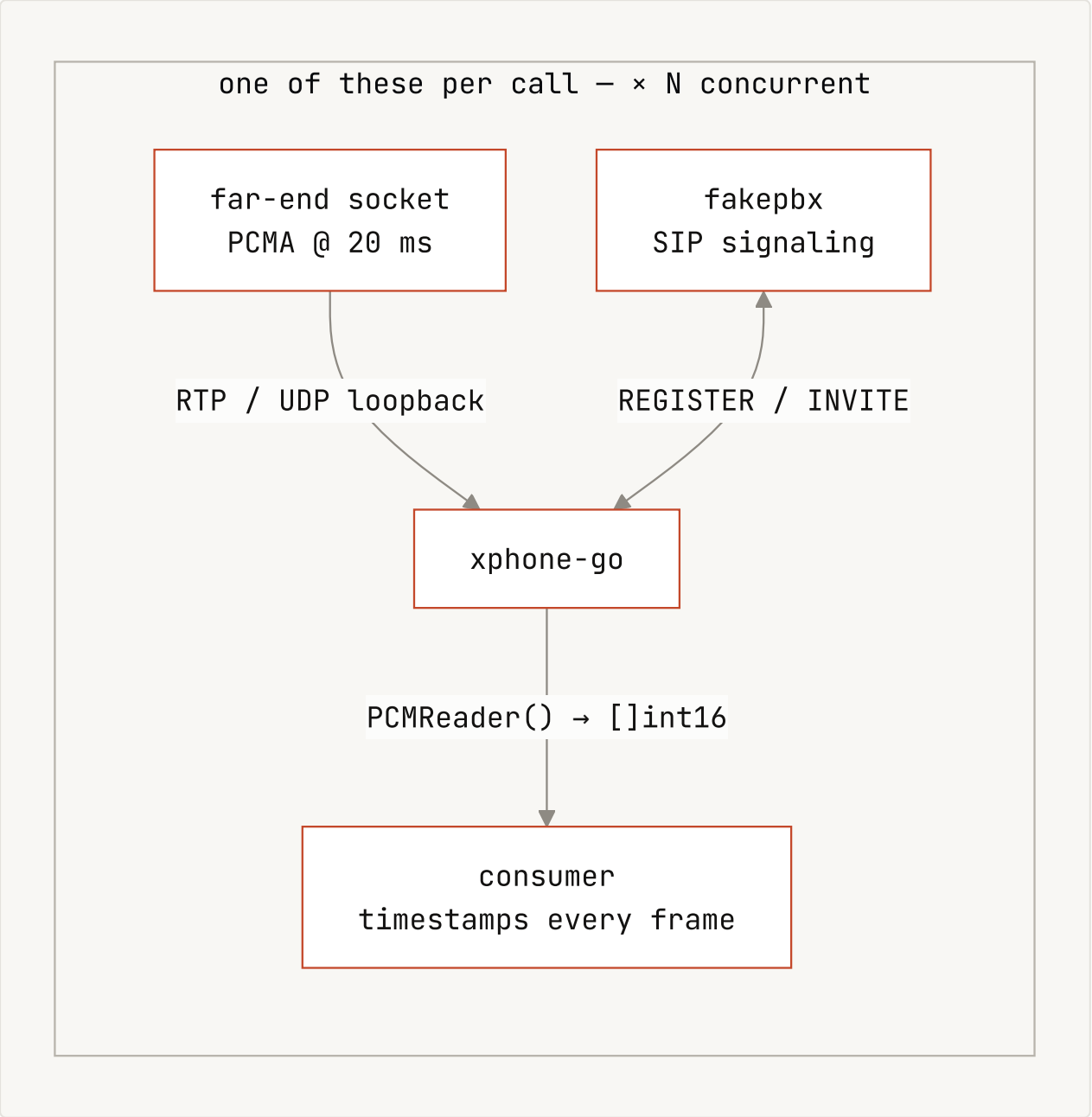
xphone normalizes inbound media into 20 ms PCM frames. For a real-time STT consumer, what matters is steady delivery: sustained stalls propagate into recognition latency, endpointing, and barge-in detection. A voice agent's perceived intelligence is downstream of its frame cadence.

So the question this article answers: **when a Go program is handling many simultaneous calls, what does the frame-arrival distribution at the application layer actually look like?**

Setup

Two experiments, honestly labeled: a **load test** over loopback, and a **live-trunk sanity check** over the real network.

The load test is reproducible on a laptop — no trunks, no Docker, no accounts. That's deliberate: fakepbx is an in-process SIP server built for exactly this kind of test, so the whole experiment is `go test` against real SIP signaling and real RTP over loopback.



- **Versions:** xphone-go v0.6.5, Go 1.26.5, macOS (Apple Silicon, `GOMAXPROCS=18`).
- **Per call:** a far-end goroutine sends 20 ms PCMA frames from its own UDP socket; the consumer reads decoded PCM frames from `call.PCMReader()` — the same channel an STT pipeline would consume — and timestamps every arrival. Note that this measures **after** xphone's jitter buffer and decoder: the pipeline is RTP → jitter buffer → decode → `PCMReader()`.

- **Load levels:** 1, 10, 25, 50 concurrent calls; ~22 s per level with the first 2 s discarded as pipeline warm-up.
- **Control:** sender-side inter-send times are recorded too. If the sender jitters, receive jitter is meaningless — every level's sender held p99 ≤ 20.05 ms with zero sends over 40 ms.

Two limitations worth stating plainly. First, senders and consumers share one machine and one scheduler; a more rigorous rig would generate RTP from a second machine over LAN. The sender control mitigates but doesn't eliminate this. Second, ~20 s per level is enough samples for a meaningful p99 at 50 calls (~50,000 frames), but **not** enough to make strong claims about maxima or rare-stall frequency — treat the Max column as anecdote, not statistics.

The live-trunk check: a single call from a Mac on residential internet (behind NAT) through a **Telnyx trunk** to a real number, measuring the voicemail greeting's audio for ~43 seconds while streaming silence upstream. One call, 2,134 frames — a sanity check that the loopback story survives contact with the real network, not a network study.

Results

Frame inter-arrival at the application layer (`PCMReader()`), per load level:

CALLS	FRAMES	P50	P95	P99	MAX	> 40 MS	IN 20 ± 5
1	1,000	20.00 ms	20.52 ms	25.02 ms	25.1 ms	0	96.2 %
10	10,000	20.00 ms	20.04 ms	24.97 ms	25.1 ms	0	99.1 %
25	24,991	20.00 ms	24.95 ms	25.01 ms	25.1 ms	0	96.7 %
50	49,971	20.00 ms	20.02 ms	24.99 ms	**65.1 ms**	0.008 %	99.3 %
Live trunk (1 call)	2,134	20.00 ms	20.02 ms	24.73 ms	**144.7 ms**	0.187 %	97.9 %

Four things worth reading out of that table:

- **The median never moves.** p50 is 20.00 ms at every load level — fifty simultaneous calls on a laptop and the typical frame is indistinguishable from a single call's.
- **The tail exists, and I'm reporting it.** At 50 calls, the worst frame in this run took 65 ms — three frame periods — and 4 frames in 50,000 crossed the 40 ms line. Small, real, and now measured instead of suspected — though a 20-second window is too short to call those numbers stable.
- **The 25 ms cluster has a likely mechanical explanation.** Across all levels, p99 lands at ~25 ms rather than drifting smoothly. xphone checks its jitter buffer both as RTP arrives and on a 5 ms drain ticker. That 5 ms cadence gives the pipeline a natural quantization boundary, which is consistent with late frames clustering around 25 ms rather than smearing continuously.
- **On this call, the real network stayed out of the application's way.** Over a live Telnyx trunk — public internet, residential NAT — the application-facing PCM cadence was indistinguishable from loopback at p50/p95/p99. That's partly the jitter buffer doing its job: it exists to absorb arrival variance, and what leaked through showed up only in the extreme tail (a 144.7 ms worst frame; ~1 frame in 500 over 40 ms).

One field note the trunk leg taught me: the first live run received **zero** frames. The harness only listened — and a listener behind NAT never opens a pinhole, so the trunk's media plane had nowhere to send audio. The fix was sending paced silence upstream. Production agents commonly send RTP early — speech, silence, or comfort media — so this rarely surfaces in a normal call flow; a receive-only benchmark has to account for it explicitly.

What surprised me

The 25 ms shelf. I expected worse under load — fifty concurrent calls on a laptop is exactly the scenario where I'd have accepted the tail getting ugly, and instead the distribution barely acknowledges the load axis at all. The p99 sits at the same ~25 ms at one call and at fifty. Going in, my mental model was "load will smear the timings"; what the data says is the distribution has a quantized tail structure that's already present at one call, and load doesn't materially move it anywhere in the range I tested.

The boring parts are the product

SIP is thirty years of accumulated edge cases: NAT traversal, codec negotiation, trunks that lie about what they support. Absorbing that entropy is the library's actual job — which is why the test matrix matters more than the feature list. Half a million calls have hardened the trunk paths I run in production: Telnyx, Twilio SIP, VoIP.ms, and Vonage. The compatibility matrix also covers the PBXes enterprises actually deploy: **Asterisk, FreeSWITCH, and 3CX**. If your company already runs a 3CX and you want an AI agent as an extension on it, that's a supported, boring Tuesday — not a research project.

Perspective

At 50 concurrent G.711 calls on this laptop, userspace Go wasn't the bottleneck. That's what this experiment shows — no more, no less: not SRTP, not Opus, not transcoding or recording or packet loss, and not what happens at several hundred calls. The next useful number is where "wasn't the bottleneck" stops being true, and finding it means longer runs, per-call breakdowns (worst per-call p99, not just the pooled distribution), and a generator on separate hardware.

The stack is open source, and so is the harness — both legs of it live in x-phone/demos (`framebench` for the loopback load test, `trunkbench` for the live-trunk check). Run them yourself, and if your numbers disagree with mine, I want to see them.

---

amenophis.dev/writing/voice-agents-real-phone-line/

amenophis.dev — Published August 7, 2026 · Amen Amouzou