vxbeamer

Release 0.0.0

Thai Pangsakulyanont

Nov 17, 2026

Guide

1 Overview **3**

2 Architecture **4**

2.1 Getting started . . . 4
Usage . . . 5
Deployment . . . 5
Backend . . . 5
2.2 Authentication . . . 6
User authentication . . . 6
Pairing a client . . . 7
API keys (personal access tokens) . . . 7
Access tokens . . . 7
Token flow . . . 7
2.3 Configuration . . . 8
Eval storage . . . 11
2.4 API . . . 11
Endpoints . . . 12
SSE events . . . 12
WebSocket protocol . . . 13
2.5 Model configurations . . . 13
Discovering them . . . 16
2.6 vxasr . . . 16
Concepts . . . 16
The `vxasr` CLI . . . 17
Replay audio at realtime . . . 17
A note on comparing models . . . 17
Development . . . 17
2.7 ASR model observations . . . 18
Fixture . . . 18

Method . . . 18
Alibaba Cloud DashScope — Qwen . . . 18
Alibaba Cloud DashScope — Qwen Omni (offline) . . . 19
Alibaba Cloud DashScope — Qwen Omni (realtime) . . . 19
The separate transcription sub-service . . . 20
With no `instructions` . . . 21
Against our ASR provider . . . 21
`qwen3.8-omni-flash-realtime` (2026-11-12) . . . 21
`qwen-audio-3.1-asr-flash-message` and `-streaming` (2026-11-12) . . . 22
Tuning the message model's parameters (2026-11-12) . . . 22
Grok Voice Transcribe 2.0 (2026-11-12) . . . 23
Scribe Realtime v2 (2026-11-14) . . . 24
iApp realtime (2026-11-14) . . . 25
Pacing: fast dump vs realtime . . . 26
BytePlus `bigmodel_nostream` (the shipped mode), fast dump vs realtime . . . 26
`gummy-realtime-v1` . . . 27
Alibaba Cloud DashScope — Fun-ASR . . . 27
BytePlus — Seed-ASR . . . 28
OpenRouter . . . 29
Meta (direct realtime API) . . . 31
Paxa Labs . . . 31
Realtime WebSocket . . . 32
Batch providers, timing compared . . . 33
Google Gemini 3.5 Transcribe . . . 33
Microsoft MAI-Transcribe-2 Streaming (Vercel AI Gateway) . . . 34
Typhoon (SCB 10X) . . . 35
Repeat runs . . . 36
`qwen3-asr-flash-realtime`, pacing compared, n=3 each . . . 36
`qwen3-asr-flash-realtime` + Groq, realtime, n=3 . . . 36
`qwen3.5-omni-flash-realtime-2026-03-15`, prompt compared, n=10 each . . . 36
Errors and timeouts observed . . . 38
Vendor-documented language support, for comparison . . . 38
Reproducing . . . 38
2.8 Backend . . . 39
Development . . . 39
2.9 Website . . . 39
Keyboard shortcuts . . . 39
Development . . . 39
2.10 vxbeamer Desktop . . . 39
Prerequisites . . . 40
Linux system libraries . . . 40
macOS . . . 40
Windows . . . 40
Development . . . 40
Building for production . . . 41
Output bundles . . . 41

CI . . . 41
2.11 vxbeamer transmitter . . . 41
Status . . . 42
Design . . . 43
Tests . . . 43
Prerequisites . . . 43
Building . . . 44
Installing . . . 44
CI . . . 44
2.12 Decisions . . . 44
A model configuration, not a provider, is the unit of selection . . . 44
Context . . . 44
Decision . . . 45
Consequences . . . 45

---

vxbeamer is a self-hosted, personal speech transcriber with a real-time web interface.

I speak into my phone. The voice message is instantly transcribed. Then I can swipe on my phone to beam the transcription to my laptop. Watch the demo video[1].

1 Overview

For most of my transcription needs, I use Google Gemini[2] (through the @lsnr[3] LINE bot) as it provides the highest accuracy. However, it comes with high latency, which makes it somewhat frustrating to use for voice typing scenarios. *(It has very high throughput though, e.g., 15 minutes of audio content can be transcribed in less than 20 seconds.)*

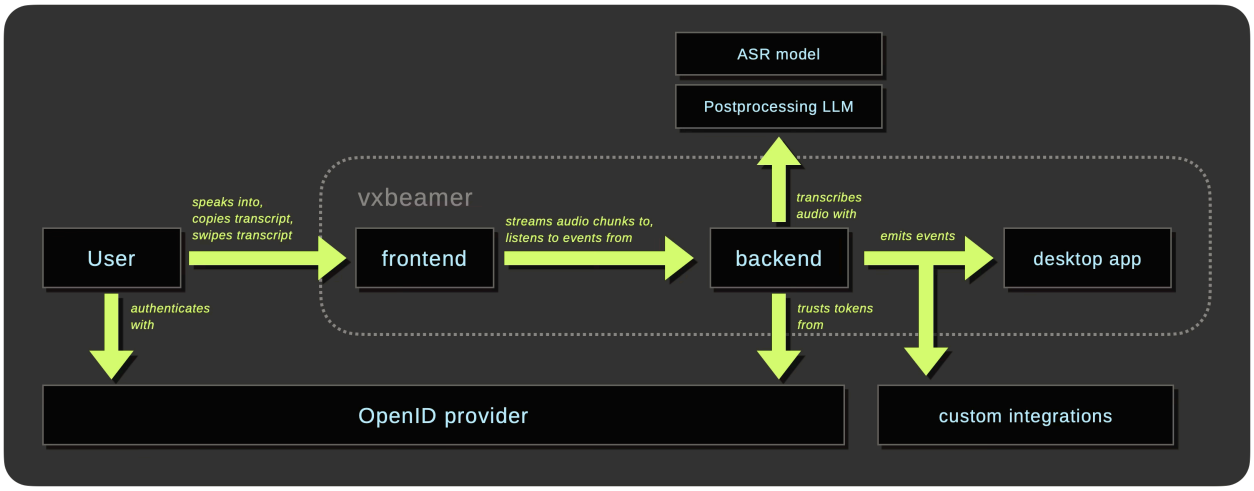
vxbeamer uses a different workflow: It uses Qwen3.5 Omni[4] Realtime[5] (via Alibaba Cloud Model Studio[6]) to transcribe speech in real-time. This trades some accuracy for significantly faster feedback.

The frontend is a PWA that can be added to the home screen. Tap the record button to transcribe, swipe right to broadcast a transcription as an event (for custom integrations), and swipe left to delete it.

This project is primarily for personal use and is not designed to be particularly flexible. That said, the setup is documented here: start with *Getting started* (page 4).

[1] https://github.com/user-attachments/assets/ff6188ed-3d5c-4d0d-bbf1-a39916091e2a
[2] https://ai.google.dev/gemini-api/docs/audio
[3] https://dt.in.th/Lsnr
[4] https://qwen.ai/blog?id=qwen3.5-omni
[5] https://www.alibabacloud.com/help/en/model-studio/realtime
[6] https://modelstudio.alibabacloud.com/

2 Architecture



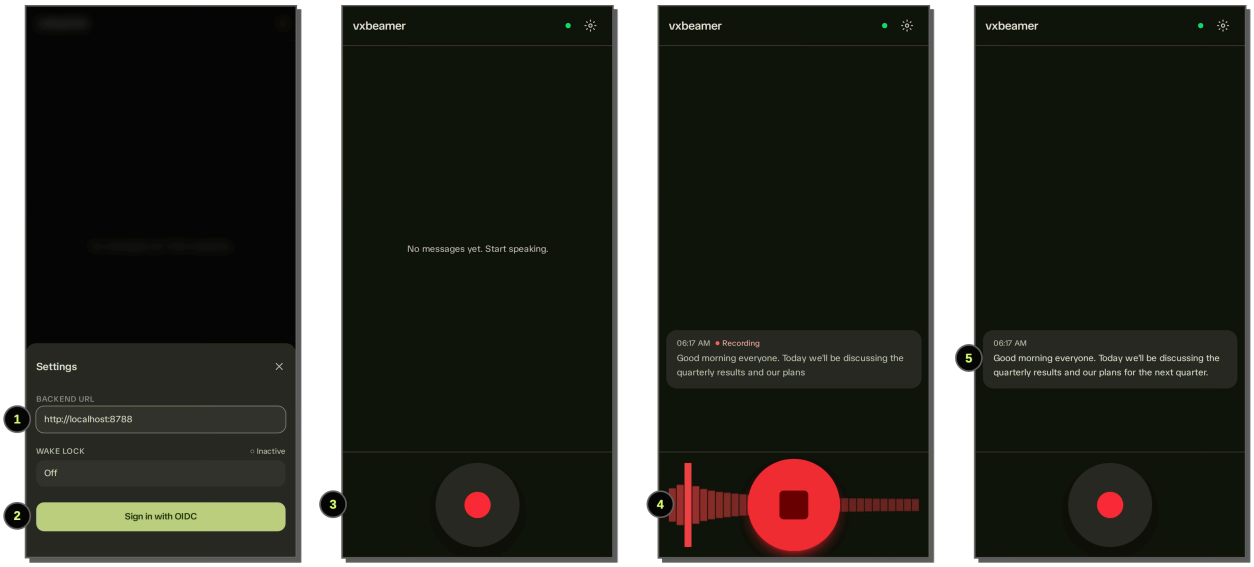
- **Frontend** — React PWA (`apps/website`), deployed statically (hosted on Vercel)
- **Backend** — Node.js/Hono server (`apps/backend`), deployed via Docker (self-hosted)
- **Desktop app** — Tauri desktop client (`apps/desktop`) that receives backend events and integrates with the local machine (basically, it's the frontend web app with extra desktop integrations)
- **ASR** — any of the providers on the *Model configurations* (page 13) page. The default is `qwen` (Qwen3-ASR-Flash Realtime via DashScope, Alibaba Cloud); the overview above describes Qwen Omni Realtime, which `ASR_PROVIDER=qwen-omni` selects
- **Post-processing** — optional; gpt-oss-120b via Groq, for configurations whose output needs formatting cleanup (not needed for Qwen Omni Realtime; the default configuration uses it when `GROQ_API_KEY` is set)

The code lives at dtinth/vxbeamer[7]. Under `apps/` are the backend, the website, the desktop app and the Android and Wear OS apps; `packages/vxasr` is the ASR client the backend uses.

2.1 Getting started

[7] https://github.com/dtinth/vxbeamer

Usage



1. Deploy the backend (see *Deployment* (page 5), below)
2. Open `/pair` on it, sign in with OIDC, and open the pairing link in the client (see *Pairing a client* (page 7))
3. To start transcribing, click the start recording button
4. To stop, click the same button
5. Click on the transcript bubble to copy, swipe left to delete, swipe right to beam to custom integrations

Deployment

A full deployment consists of three parts:

1. **OIDC provider** — an OIDC-compatible identity provider such as Authentik (see *Authentication* (page 6)).
2. **Backend** — a self-hosted server you run yourself (see below).
3. **Frontend** — the web application, available at vxbeamer.vercel.app[8]. This hosted instance is provided as-is and connects to whichever backend you pair it with: it learns the backend's address from the pairing link, so there is no URL to type. Only the frontend is hosted; you must run your own backend.

Backend

The backend is distributed as a Docker image.

```
services:
  backend:
    image: ghcr.io/dtinth/vxbeamer:latest
    pull_policy: always
    restart: unless-stopped
    expose:
```

(continues on next page)

[8] https://vxbeamer.vercel.app

(continued from previous page)

```
    - 8787
  environment:
    - DASHSCOPE_API_KEY
    - GROQ_API_KEY
    - BYTEPLUS_API_KEY
    - OPENAI_API_KEY
    - XAI_API_KEY
    - ELEVENLABS_API_KEY
    - IAPP_API_KEY
    - OPENROUTER_API_KEY
    - META_API_KEY
    - PAXA_API_KEY
    - GEMINI_API_KEY
    - AI_GATEWAY_API_KEY
    - OIDC_DISCOVERY_URL
    - OIDC_CLIENT_ID
    - OIDC_SECRET
    - API_KEYS
```

The `environment` list above is only the common subset: every variable is listed on the *Configuration* (page 8) page.

2.2 Authentication

vxbeamer comes with two authentication methods:

- **User authentication** via OIDC, for interactive use from the frontend.
- **API keys** (via `API_KEYS`), for integration with scripts.

User authentication

The OIDC provider must support:

1. **PKCE flow** — the frontend performs the authorization code flow with PKCE and exchanges the resulting ID token for a session with the backend.
2. **Discovery document** — the provider must expose its configuration at the standard `/.well-known/openid-configuration` path.
3. **CORS** — cross-origin requests must be allowed, since the frontend will contact the provider directly from the browser.
4. **Restricted token issuance** — the provider must only issue ID tokens to authorized users. There is no built-in user whitelist in vxbeamer itself, so access control must be enforced at the provider level.

Authentik[9] is a self-hosted, open-source identity provider that meets all of these requirements.

[9] https://goauthentik.io

Pairing a client

Clients that cannot easily sign in themselves (a PWA on iOS, the Android app, anything on a different machine from the browser you are signed in on) pair instead. Open `/pair` on your backend, sign in, and it shows a one-time pairing link and QR code. Open the link in the client, scan the QR code, or paste the link into its pairing box. The link carries the backend's address and a 12-character code; the client redeems the code at `POST /auth/pair` for its own session.

Codes are valid for two minutes and work once. They are kept in memory, so a backend running several instances would need to share them.

The `/pair` page runs the OIDC code flow itself, so the provider must allow `https://<your-backend>/pair` as a redirect URI. Register it as an exact match, not a wildcard. Set `PUBLIC_URL` if the backend sits behind a proxy and the link should not use the address it sees.

API keys (personal access tokens)

Set the `API_KEYS` environment variable in the format `<sub>:<key>`. In case of multiple keys, separate by commas:

```
API_KEYS=your-sub-claim:my-secret-key,your-sub-claim:another-secret
```

To find your `sub` claim:

1. Pair the web app with your backend (see *Pairing a client* (page 7))
2. Open Settings (icon)
3. Copy your `sub` from the "Signed in" section

The `<sub>` must match the `sub` claim of your OIDC user. You can have multiple keys for the same user (useful for key rotation or different integrations).

API keys are not used directly for authenticated requests. Instead, scripts exchange them for short-lived access tokens via `POST /auth/token`. This keeps all authentication consistent: protected endpoints only accept session tokens, regardless of whether they came from OIDC or API key exchange.

Access tokens

All protected endpoints require an access token. Pass it as:

- `Authorization: Bearer <token>` header, or
- `?access_token=<token>` query parameter

Token flow

For OIDC users (interactive frontend):

1. Exchange OIDC id_token for access & refresh tokens: `POST /auth/session` with `id_token`
2. Access token (15 min TTL) is used for protected endpoints

3. When access token < 10 minutes remaining, refresh both tokens: `POST /auth/refresh` with `refresh_token`
4. Refresh token is valid for 3 days from the last refresh

For API keys (scripts/integrations):

1. Exchange API key for access token: `POST /auth/token` with `api_key`
2. Access token (15 min TTL) is used for protected endpoints
3. No refresh token is issued; obtain a new access token by exchanging the API key again

2.3 Configuration

The backend is configured with environment variables.

Variable	Required	Description
`DASHSCOPE_API_KEY`	Default provider	Alibaba Cloud DashScope key for Qwen3-ASR-Flash, Qwen Omni Realtime and Qwen-Audio ASR. The default provider is `qwen`, so it is needed unless `ASR_PROVIDER` names another one (`mock` needs no key)
`DASHSCOPE_WORKSPACE_ID`	No	Model Studio workspace id. Needed only by `qwen-omni/qwen3.8-omni-flash-realtime` and the `qwen-audio/*` models, which are served from `<id>.ap-southeast-1.maas.aliyuncs.com`
`API_KEYS`	No	Comma-separated `sub:secret` pairs for API key exchange
`GROQ_API_KEY`	No	Groq API key for gpt-oss-120b post-processing; enables the `+groq` configurations
`ASR_CONFIGURATION`	No	Default configuration id (default: derived from `ASR_PROVIDER/ASR_MODEL/GROQ_API_KEY`)
`ASR_CONFIGURATIONS`	No	Comma-separated configurations clients may select (default: every configuration with credentials)

continues on next page

Table 1 – continued from previous page

Variable	Required	Description
ASR_PROVIDER	No	Provider for the derived default: qwen (default), qwen-omni, qwen-audio, byteplus, openai, xai, elevenlabs, iapp, openrouter, meta, paxa, gemini, ai-gateway, or mock
ASR_MODEL	No	Model for the derived default (default: the provider's own default model)
BYTEPLUS_API_KEY	No	BytePlus key; enables the byteplus configurations
BYTEPLUS_LANGUAGE	No	BytePlus language hint, e.g. th-TH (default: unset — Mandarin/English only)
BYTEPLUS_RESOURCE_ID	No	BytePlus resource id (default: volc.seedasr.sauc.duration)
BYTEPLUS_BASE_URL	No	BytePlus endpoint base, without the mode path segment
OPENAI_API_KEY	No	OpenAI key; enables the openai (gpt-live-transcribe) configuration
XAI_API_KEY	No	xAI key; enables the xai (grok-voice-transcribe-2.0, streaming) configuration
ELEVENLABS_API_KEY	No	ElevenLabs key; enables the elevenlabs (scribe_v2_realtime) configuration
IAPP_API_KEY	No	iApp key; enables the iapp (chinda-asr-realtime-v1, streaming, Thai only) configuration
OPENROUTER_API_KEY	No	OpenRouter key; enables the openrouter configurations (microsoft/mai-transcribe-2, meta/muse-voice-transcribe-1.0, microsoft/mai-transcribe-1.5, google/gemini-3.5-transcribe)

continues on next page

Table 1 – continued from previous page

Variable	Required	Description
META_API_KEY	No	Meta key; enables the `meta` (`muse-voice-transcribe-1.0`, direct realtime API) configuration
PAXA_API_KEY	No	Paxa Labs key; enables the `paxa` configurations (batch and realtime)
PAXA_CONVENTION	No	`written` to render numbers as digits and use the Thai repetition mark (default: the vendor's `spoken`)
GEMINI_API_KEY	No	Google Gemini API key; enables the `gemini` (`gemini-3.5-transcribe-live`) configuration
AI_GATEWAY_API_KEY	No	Vercel AI Gateway key; enables the `ai-gateway` (`microsoft/mai-transcribe-2-streaming`) configuration
OIDC_DISCOVERY_URL	No	OIDC provider discovery URL (alternative to API keys)
OIDC_CLIENT_ID	No	OIDC client ID (default: `vxbeamer-mobile`)
OIDC_AUDIENCE	No	Expected token audience (default: same as client ID)
OIDC_SECRET	No	HMAC secret for session tokens (default: `local-dev-secret`)
PUBLIC_URL	No	Address clients should use for pairing links (default: the address the request came in on)
WEBHOOK_URL	No	Endpoint to POST completed transcriptions to
PORT	No	HTTP port (default: 8787)
WS_IDLE_TIMEOUT_MS	No	How long a recording socket may go without audio before it is closed and its vendor connection released (default: `60000`; `0` disables)
QWEN_OMNI_STICKY_LINGER_MS	No	How long an unused Qwen Omni connection is kept open after a recording, for the next one from the same client (default: `30000`)

continues on next page

Table 1 – continued from previous page

Variable	Required	Description
QWEN_OMNI_STICKY_MAX_AUDIO_S	No	Audio a kept Qwen Omni connection may carry across recordings before it is retired (default: 180)

Eval storage

Where eval **votes** and **eval-sets** are saved. The backend only signs upload URLs; the browser uploads directly to the bucket, so recordings never pass through the server and no bucket credential ever reaches a browser. Setting EVAL_STORAGE_BUCKET turns the feature on; without it, evals still run and winners still apply, they are just not recorded.

Variable	Required	Description
EVAL_STORAGE_BUCKET	No	S3-compatible bucket for votes and eval-sets. Enables eval storage
EVAL_STORAGE_ACCESS_KEY_ID	With bucket	Access key id used to sign upload URLs
EVAL_STORAGE_SECRET_ACCESS_	With bucket	Secret access key used to sign upload URLs
EVAL_STORAGE_REGION	No	Bucket region (default: us-east-1)
EVAL_STORAGE_ENDPOINT	No	Custom S3-compatible endpoint, e.g. MinIO or R2 (default: AWS)
EVAL_STORAGE_FORCE_PATH_STY	No	Set to true for endpoints needing path-style addressing, e.g. MinIO
EVAL_STORAGE_PREFIX	No	Key prefix placed before votes/ and eval-sets/

Objects are keyed by kind, then UTC date, then time and message id:

```
votes/2026/07/16/2026-07-16T09-30-00-123Z-<messageId>.json
eval-sets/2026/07/16/2026-07-16T09-30-00-123Z-<messageId>.json
```

A **vote** ({"type":"vote",…}) is written on every winner pick and carries the winning configuration id, the full candidate list, per-candidate cost/latency/errors, and the audio's duration — but **no audio and no transcripts**, so it is safe to store unconditionally. An **eval-set** ({"type":"eval-set",…}) is written only when the user ticks save-for-eval, and adds the recording as a base64 WAV plus every candidate's transcript. A pick's vote and eval-set share a key suffix, so one finds the other.

2.4 API

The backend exposes a REST + SSE + WebSocket API on port 8787 (PORT). All endpoints require an access token, obtained by exchanging OIDC id_tokens, API keys or a pairing code, except /healthz, /auth/* and the /pair page.

Endpoints

Method	Path	Description
GET	/healthz	Health check
GET	/auth/config	OIDC configuration for the frontend
POST	/auth/session	Exchange an OIDC id_token for access & refresh tokens
POST	/auth/token	Exchange an API key for an access token (no refresh)
POST	/auth/refresh	Exchange a refresh token for new access & refresh tokens
POST	/auth/pair	Redeem a pairing code for a session (rate-limited)
GET	/pair	The pairing page: signs in with OIDC and shows a pairing link and QR code
POST	/pair/codes	Mint a one-time pairing code, link and QR code
GET	/sse	Server-Sent Events stream of all activity
GET	/asr/configurations	List the selectable model configurations
GET	/messages	List all messages (last 24 hours)
GET	/messages/:id	Get a single message
DELETE	/messages/:id	Delete a message
POST	/messages/:id/swipe	Broadcast a swipe event for integrators
POST	/messages/:id/winner	Replace the primary answer with an eval winner's
WebSocket	/ws	Stream PCM audio for transcription
WebSocket	/asr/eval	Transcribe without storing a message, for comparing configurations

POST /pair/codes returns { code, link, svg, expires_in }. POST /auth/pair takes { code } and answers 401 for an invalid or expired code and 429 when rate-limited; see *Pairing a client* (page 7).

POST /messages/:id/winner takes { configurationId, transcript } and replies with { ok: true, upload }. When eval storage is configured, upload carries presigned PUT URLs for this pick's vote and eval-set; it is null otherwise. Signing is offline, so the URLs come back with the winner rather than costing a second round-trip — and because the winner is applied before they are minted, storage being unavailable can never fail the pick.

SSE events

Connect to /sse to receive real-time events. Pass ?events=<type> to filter, e.g. ?events=swiped to receive only swipe events (the initial snapshot is skipped when a filter is active).

Event type	Description
`snapshot`	Initial state — all current messages
`created`	A new recording session started
`updated`	Transcript updated (partial or final)
`deleted`	A message was deleted
`swiped`	A message was swiped right (`eventId` is unique per swipe event)

WebSocket protocol

Connect to `/ws?access_token=<token>`. Send raw PCM audio as binary frames (16 kHz, 16-bit signed, mono, little-endian). Send `{ "type": "stop" }` as a text frame to end the session gracefully.

Optionally add `?configuration=<id>` to transcribe with a specific **model configuration** instead of the default one. Omit it and the session uses the configured default.

Two more optional query parameters: `reference_id`, an id of your choosing that is stored on the message as `referenceId`, and `client_id`, an id a client makes up for each launch, which lets a provider that supports it (currently `qwen-omni`) keep a vendor connection open between recordings from the same client. A socket that sends no audio for `WS_IDLE_TIMEOUT_MS` is closed with code `1008`.

The `/asr/eval` WebSocket takes the same audio frames and the same `access_token` and `configuration` parameters, but stores no message: the web app opens one per configuration to replay a recording through each when comparing models.

The model configurations a client can select are on the *Model configurations* (page 13) page. Tokens are explained on the *Authentication* (page 6) page.

2.5 Model configurations

The raw results behind the notes below are in the *vendor notes* (page 18); `vxasr` is described on its own *page* (page 16).

A configuration is a provider, a model, and the post-processing chain applied to it. Post-processing is part of a configuration's identity rather than a request flag, so a model raw and the same model with Groq formatting are two separately selectable configurations that can be compared on equal terms:

Configuration id	Needs
qwen/qwen3-asr-flash-realtime-2025-10-27	DASHSCOPE_API_KEY
qwen/qwen3-asr-flash-realtime-2025-10-27+groq	DASHSCOPE_API_KEY, GROQ_API_KEY
qwen/qwen3-asr-flash-realtime-2026-02-10	DASHSCOPE_API_KEY
qwen/qwen3-asr-flash-realtime-2026-02-10+groq	DASHSCOPE_API_KEY, GROQ_API_KEY
qwen-omni/qwen3.5-omni-flash-realtime-2026-03-	DASHSCOPE_API_KEY
qwen-omni/qwen3.5-omni-plus-realtime-2026-03-1	DASHSCOPE_API_KEY
qwen-omni/qwen3-omni-flash-realtime-2025-12-01	DASHSCOPE_API_KEY
qwen-omni/qwen3.8-omni-flash-realtime	DASHSCOPE_API_KEY, DASHSCOPE_WORKSPACE_ID
qwen-audio/qwen-audio-3.1-asr-flash-message	DASHSCOPE_API_KEY, DASHSCOPE_WORKSPACE_ID
qwen-audio/qwen-audio-3.1-asr-flash-streaming	DASHSCOPE_API_KEY, DASHSCOPE_WORKSPACE_ID
byteplus/bigmodel_nostream	BYTEPLUS_API_KEY
byteplus/bigmodel_nostream+groq	BYTEPLUS_API_KEY, GROQ_API_KEY
openai/gpt-live-transcribe	OPENAI_API_KEY
xai/grok-voice-transcribe-2.0	XAI_API_KEY
elevenlabs/scribe_v2_realtime	ELEVENLABS_API_KEY
iapp/chinda-asr-realtime-v1	IAPP_API_KEY
openrouter/microsoft/mai-transcribe-2	OPENROUTER_API_KEY
openrouter/meta/muse-voice-transcribe-1.0	OPENROUTER_API_KEY
openrouter/microsoft/mai-transcribe-1.5	OPENROUTER_API_KEY
openrouter/google/gemini-3.5-transcribe	OPENROUTER_API_KEY
paxa/paxa-stt-lite-v1-preview	PAXA_API_KEY
paxa/paxa-stt-lite-realtime-v1-preview	PAXA_API_KEY
meta/muse-voice-transcribe-1.0	META_API_KEY
gemini/gemini-3.5-transcribe-live	GEMINI_API_KEY
ai-gateway/microsoft/mai-transcribe-2-streaming	AI_GATEWAY_API_KEY
mock/mock	nothing

Most Qwen model ids are dated snapshots, not floating ones — the vendor repoints undated ids without notice, which would make a vote name a moving target. The exceptions are `qwen-omni/qwen3.8-omni-flash-realtime` and the two `qwen-audio/*` ids, which the vendor lists with no snapshot id yet. The Qwen Omni models need no `+groq` variant: their output is already well-formatted (Thai words in Thai, product names in Latin), which is what Groq formatting was tidying up for the plain ASR models — running it on top of Omni's output measurably added nothing.

BytePlus exposes two modes at two endpoints, and only `bigmodel_nostream` accepts a language — the bidirectional `bigmodel` mode covers Mandarin, English and a few Chinese dialects and nothing else, so on any other language it returns confident nonsense. Only `bigmodel_nostream` is offered as a configuration for that reason; the provider still serves `bigmodel` for deployments in those languages. Set `BYTEPLUS_LANGUAGE` (e.g. `th-TH`) or BytePlus falls back to that same Chinese/English default set. Note that `bigmodel_nostream` returns results only after 15 s of audio or the final packet, so it emits few or no partials — it is accuracy-tuned rather than low-latency.

OpenRouter is a batch HTTP endpoint fanning out to many backing vendors under one key, not a live socket like every other provider here — the whole clip is sent as one file once recording stops, so it produces no partial

transcript. Four models are declared, each tried live on the same fixture `testdata/OBSERVATIONS.md` uses. `microsoft/mai-transcribe-2` is the default: when the three were finally timed against each other rather than compared on transcript alone, it was the fastest and the cheapest, and its transcript matched the best of them. `meta/muse-voice-transcribe-1.0` held the default before that on transcript quality, and turned out to be four times slower than the others; `microsoft/mai-transcribe-1.5` was the original pick from a comparison of 19 models. `google/gemini-3.5-transcribe` is the Gemini model described below, called as a batch request.

The `meta` provider speaks to that same `muse-voice-transcribe-1.0` model directly over Meta's own realtime WebSocket endpoint instead of through OpenRouter's batch wrapper, so it is the only configuration of this model that actually streams partial transcripts. Live-tried against the same fixture (`testdata/OBSERVATIONS.md`): the transcript came back slightly less accurate than the OpenRouter route for this same model on that one clip, and usage is billed in whole seconds rounded up rather than OpenRouter's fractional cost — a genuine tradeoff, not a strict upgrade, which is why both configurations are kept.

Paxa Labs is a batch endpoint too, but takes JSON with the audio base64-encoded rather than a multipart upload. It is the only model compared in `testdata/OBSERVATIONS.md` that needed no instruction to behave: no filler tags, no spaces inserted between Thai words, no English translated into Thai, and byte-identical output across repeated runs of every clip. It costs roughly four times `mai-transcribe-2` for near-identical speed, so it is offered alongside rather than as a default. `PAXA_CONVENTION=written` renders numbers as digits and uses the Thai repetition mark; the vendor's default spells them as spoken.

`paxa/paxa-stt-lite-realtime-v1-preview` is Paxa's realtime WebSocket. The vendor detects the end of speech, so the final transcript arrives about 55 ms after the recording ends. It costs 1.5 times the batch rate, and the whole connection is charged, including silence.

Gemini 3.5 Transcribe is offered two ways. Through OpenRouter it is a batch call: accurate and cheap, but 4–6 s per request whatever the clip length. The `gemini` provider uses the Live model instead, in push-to-talk mode with smart transcription (fillers and false starts removed): partial transcripts about every 0.5 s, and the final 0.3–0.6 s after the audio ends. A Live session can stream for 10 minutes at most. In tests on clips joined from separate recordings, push-to-talk with smart transcription sometimes dropped a later sentence; see `testdata/OBSERVATIONS.md`.

`ai-gateway/microsoft/mai-transcribe-2-streaming` is the streaming version of MAI-Transcribe-2, offered only through the Vercel AI Gateway, and reached with the AI SDK's `experimental_streamTranscribe`. Partial transcripts come about every 0.3 s, and the final 0.33–0.49 s after the audio ends. On clips with pauses between sentences it kept every sentence. It costs $0.54 per hour of audio.

Ids contain `+`, which decodes to a space in a query string, so clients must URL-encode them — `URLSearchParams` does this automatically. Several ids also contain `/` (for example `openrouter/microsoft/mai-transcribe-1.5`), which needs no special handling in a query string value.

By default a configuration is selectable once the environment carries every credential it needs — which means `mock/mock`, needing none, is always selectable. Set `ASR_CONFIGURATIONS` to pin the list down explicitly. Requests naming an unknown or disabled configuration are closed with code `1008`, and one named without its credentials with `1011` (a server misconfiguration), before any message is created.

Discovering them

`GET /asr/configurations` returns the selectable set, so a client does not have to hardcode the table above:

```
{
  "primaryConfigurationId": "qwen-omni/qwen3.5-omni-flash-realtime-2026-03-15",
  "configurations": [
    {
      "id": "qwen-omni/qwen3.5-omni-flash-realtime-2026-03-15",
      "label": "Qwen3.5-Omni-Flash Realtime (2026-03-15, raw)",
      "providerId": "qwen-omni",
      "model": "qwen3.5-omni-flash-realtime-2026-03-15",
      "postProcessing": [],
      "configured": true
    }
  ]
}
```

Every listed `id` is one `/ws` accepts in `?configuration=`, which is what makes this list usable as a fan-out plan: an eval opens one socket per entry. `primaryConfigurationId` names the configuration the primary path uses; it is listed alongside the rest, since it competes on the same terms.

`configured` is almost always true — a configuration normally earns its place here by being credentialled. It is false only where an `ASR_CONFIGURATIONS` entry (or the default, which is always listed) was named without its credentials; such an entry is shown rather than hidden so the gap is visible, but connecting to it would close immediately. The response never carries credentials — not their values, and not the names of the missing variables either; the `/ws` close reason is where an operator reads which one to set.

2.6 vxasr

ASR client for vxbeamer[10], a self-hosted personal speech transcriber.

Wraps several speech-recognition vendors behind one streaming interface, so they can be swapped and compared without the calling code knowing which is which.

Concepts

`ASRProvider` — a vendor+model behind a streaming session: `createSession(callbacks)` returns something you feed `sendAudio(chunk)` and then `finish()`. Callbacks report `onPartial` / `onFinal` / `onUsage` / `onEnd` / `onError`.

Model configuration — a provider, a model, and its post-processing chain. This, not the provider, is the unit you select and compare: `qwen/qwen3-asr-flash-realtime` and `qwen/qwen3-asr-flash-realtime+groq` are two different things to evaluate, and they compete on equal terms.

Audio format — 16 kHz, 16-bit signed little-endian, mono. 32000 bytes per second. This is what the browser's AudioWorklet captures, so nothing needs resampling anywhere in the pipeline.

[10] https://github.com/dtinth/vxbeamer

The vxasr CLI

Runs one configuration against an audio file — no server, no browser, no microphone. This is the fastest way to check that a newly wired adapter actually works.

```
# What can I run, and do I have the credentials?
vxasr --list

# Transcribe a file (credentials come from the environment)
node --env-file=.env vxasr qwen/qwen3-asr-flash-realtime testdata/test-audio.bin
```

```
qwen/qwen3-asr-flash-realtime — 9.22s of audio (raw), realtime
project □□□□□□□□□□□□□□□□□ TypeScript □□□ framework □□□□ Elysia □□□ deploy □□□□□□
 ↪Railway□

84 partials in 9.46s
  dashscope:qwen3-asr-flash:seconds: 10 □ $0.000035 = $0.000350
  total: $0.000350
```

Accepts a WAV or headerless raw PCM. WAV is detected by content rather than file extension, and its header is stripped — but the audio inside must already be 16 kHz/16-bit/mono, since resampling it here would mean testing something other than what the app sends:

```
ffmpeg -i input.mp3 -ar 16000 -ac 1 -c:a pcm_s16le output.wav
```

The transcript goes to stdout and diagnostics to stderr, so it pipes cleanly. Exits non-zero when the configuration is unknown, lacks credentials, or errors.

Replay audio at realtime

The CLI paces audio at 100 ms per chunk by default, as a live recording would. **`--fast` is a probe, not an optimisation.** Feeding audio faster than realtime does not reproduce live behaviour: BytePlus hangs outright and never returns a result, and vendors document a 100–200 ms send interval. Billing is per audio-second rather than per connection-second, so realtime pacing costs nothing extra.

A note on comparing models

These models are LLMs with audio intake, so they are **not deterministic** — the same audio can transcribe differently across runs. A single run is one sample, not a measurement. Judge a model over many recordings rather than one.

Development

```
vp install    # install dependencies
vp test       # run the unit tests
vp check      # format, lint, type check
vp pack       # build the library and CLI
```

Unit tests use the `mock` provider and never make network calls, so they need no credentials and cost nothing.

2.7 ASR model observations

Raw results from running models against `test-audio.bin`. **Observations only** — what was sent, what came back. No rankings.

Everything here was run against live vendor endpoints on **2026-07-16**.

Fixture

`test-audio.bin` — raw PCM, 16 kHz / 16-bit / mono, little-endian, no header. 294,986 bytes = 9.218 s. Thai speech containing English technical terms (“TypeScript”, “Elysia”, “Railway”, “MongoDB Atlas”).

Method

- Audio sent at **realtime pacing**: 3200-byte chunks (100 ms), 100 ms apart, unless a row says otherwise.
- **One run per row unless a repeat count is given.** These models are LLMs with audio intake and are **not deterministic** — see *Repeat runs* (page 36), where identical inputs produced different outputs.
- Transcripts are reproduced **exactly**, including case, spacing and trailing punctuation.

Alibaba Cloud DashScope — Qwen

OpenAI-compatible realtime protocol. `wss://dashscope-intl.aliyuncs.com/api-ws/v1/realtime?model=<id>`, `Authorization: Bearer`, `session.update` with `turn_detection: null` (Manual mode), audio base64 in `input_audio_buffer.append`.

model	output
`qwen3-asr-flash-re` (floating)	`project □□□□□□□□□□□□□□□□□ TypeScript □□□ framework □□□□ Elysia □□□ deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□ □□□□□□□□□.`
`qwen3-asr-flash-re`	`project □□□□□□□□□□□□□□□□□ TypeScript □□□ framework □□□□ Elysia □□□ deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□ □□□□□□□□□.`
`qwen3-asr-flash-re`	`□□□□□□□□□□□□□□□□□□□□□□□□□□ typescript □□□ framework □□□□ elixir □□□ deploy □□□□□ railway □□□□□□ mongodb atlas □□□□□□□□□□□□□□□□□□□□ □□□□□□`
`qwen3-asr-flash-re` + Groq	`project □□□□□□□□□□□□□□□□□ TypeScript □□□ framework □□□□ Elysia □□□ deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□ □□□□□□□□□.`
`qwen3-omni-flash-r`	**no result** — 60 s timeout. It speaks its own protocol, not this one; see the Qwen Omni (realtime) section.

Usage reported: `dashscope:qwen3-asr-flash:seconds` quantity **10**, unit price `$0.000035` → `$0.000350`. With Groq, additionally: `groq:openai/gpt-oss-120b:input-tokens` **224** @ `$1.5e-7`, `output-tokens` **~160** @ `$6e-7` → `$0.000479` total.

Alibaba Cloud DashScope — Qwen Omni (offline)

Not the realtime WS protocol — these are offline/batch models reached over HTTP. POST `https://dashscope-intl.aliyuncs.com/compatible-mode/v1/chat/completions` (OpenAI-compatible), `Authorization: Bearer`, a `messages` array carrying an `input_audio` content part (base64 WAV data URI) alongside a text prompt, with `modalities: ["text"]`.

The whole 9.218 s clip is uploaded at once; there is no pacing.

Prompt used for every row below — **these models are prompt-driven, so the prompt is part of the input**:

    Transcribe this audio verbatim. Output only the transcript.

model	output	wall
qwen3.5-omni-plu:	□□□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□□□□□□ Elysia □□□ deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□□□□□□□□□	1.7 s
qwen3-omni·	□□□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□□□□□□ Elysia □□□ Deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□□□□□□□□□	1.7 s
qwen3.5-omni-fla:	□□□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□□□□□□ Alecia □□□ deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□□□□□□□□□	1.7 s

Billing is in tokens, not seconds — a different cost model from every ASR model above:

model	prompt (audio + text)	completion	total
`qwen3.5-omni-plus`	89 (65 + 24)	32	121
`qwen3.5-omni-flash`	89 (65 + 24)	31	120
`qwen3-omni-flash`	137 (117 + 20)	55	192

The same audio tokenises differently across the two generations (65 vs 117 audio tokens).

The omni **realtime** models transcribe from their own weights too, over their own protocol — see the Qwen Omni (realtime) section. These offline rows are a different endpoint (HTTP, not WS) reached with a prompt rather than an `instructions` string.

Alibaba Cloud DashScope — Qwen Omni (realtime)

These have **their own protocol** — not the Qwen ASR realtime one. Same URL shape (`wss://□/api-ws/v1/realtime?model=<id>`; the vendor now recommends the workspace domain `{WorkspaceId}.ap-southeast-1.maas.aliyuncs.com`, though the legacy host answered), `Authorization: Bearer`, but:

- `session.update` with `modalities: ["text"]`, `input_audio_format: "pcm"`, `turn_detection: null` (manual), and an `instructions` string.
- `input_audio_buffer.append` (base64) → `input_audio_buffer.commit` → `response.create`.
- The model's own output arrives on `response.text.delta` / `response.text.done`.
- No `session.finish` — that is the ASR protocol's message, not this one's.

Instructions used for every row below — **these models are instruction-driven, so the instruction is part of the input**:

```
Transcribe the user's audio verbatim. Output only the transcript, nothing else.
```

model	output
qwen3.5-omni-plus-rea	▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ TypeScript ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ Elysia ▯▯▯ deploy ▯▯▯▯▯ Railway ▯▯▯▯▯▯ MongoDB Atlas ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
qwen3.5-omni-flash-re	▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ TypeScript ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ Elysia ▯▯▯ deploy ▯▯▯▯▯ Railway ▯▯▯▯▯▯ MongoDB Atlas ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯
qwen3-omni-flas	Project ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ TypeScript ▯▯▯ framework ▯▯▯▯ Elysia ▯▯▯ deploy ▯▯▯▯▯ Railway ▯▯▯▯▯▯ MongoDB Atlas ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯

Billing is in **tokens**:

model	input (text + audio)	output	total
`qwen3.5-omni-plus-realtime`	114 (44 + 70)	32	146
`qwen3.5-omni-flash-realtime`	114 (44 + 70)	32	146
`qwen3-omni-flash-realtime`	165 (40 + 125)	47	212

The vendor documents speech recognition for **113 languages and dialects**, a 120-minute session cap, and per-model conversation-history limits (`qwen3.5-omni-plus-realtime` 100 audio turns / 600 s; `-flash-realtime` 80 / 480 s; `qwen3-omni-flash-realtime` 8 turns).

The separate transcription sub-service

Independently of the model's own output, a session may name an ASR model in `session.input_audio_transcription`; its results arrive on `conversation.item.input_audio_transcription.*`. **This is not the omni model transcribing** — it is a different model running alongside:

`session.input_audio_transcripti`	`conversation.item.input_audio_transcription.completed`
`{ model: "gummy-realtime-v1" }`	project ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ TypeScript ▯▯▯ framework ▯▯▯▯ Elysia ▯▯▯ deploy ▯▯▯▯▯ Railway ▯▯▯▯▯▯ MongoDB Atlas ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯.
`{}` (no model named)	no event at all
`{ model: "qwen3-asr-flash-realti }`	project ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯ TypeScript ▯▯▯ framework ▯▯▯▯ Elysia ▯▯▯ deploy ▯▯▯▯▯ Railway ▯▯▯▯▯▯ MongoDB Atlas ▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯.

Note these differ from the omni model's own `response.text` output above: this stream renders `project` in Latin, the omni model renders ▯▯▯▯▯▯▯▯ in Thai.

With no `instructions`

Sending `response.create` without an `instructions` string produced a **conversational reply**, in Thai, offering to help design the project described in the clip — a Tech Stack breakdown with numbered sections and follow-up questions. Not a transcript. The audio's content drove a conversation.

Observed event sequence: `session.created`, `session.updated`, `conversation.item.input_audio_transcription.delta`, `input_audio_buffer.committed`, `response.created`, `response.output_item.added`, `conversation.item.created`, `response.content_part.added`, `response.text.delta`, `conversation.item.input_audio_transcription.completed`, `response.text.done`, `response.content_part.done`, `response.output_item.done`, `response.done`.

Against our ASR provider

Our `qwen` provider speaks the ASR-specific protocol (`session.finish`) and waits for ASR events, so it does not drive these models:

model	result
`qwen3-omni-flash-realtime`	no result, 60 s timeout
`qwen3.5-omni-plus-realtime`	no result, 60 s timeout
`qwen3.5-omni-flash-realtime`	no result, 60 s timeout

`qwen3.8-omni-flash-realtime` (2026-11-12)

Served only from a workspace endpoint: `wss://<WorkspaceId>.ap-southeast-1.maas.aliyuncs.com/api-ws/v1/realtime?model=qwen3.8-omni-flash-realtime`, not the shared `dashscope-intl` one. No dated snapshot is listed in the vendor's docs. The vendor's page describes a new nested `session.update` shape (`audio.input.format`, `audio.output`), but our existing flat handshake (`input_audio_format: "pcm"`, `sample_rate`, `turn_detection: null`) is accepted unchanged, so the same adapter drives it.

On the 9.2 s fixture, four realtime runs: two returned the same transcript as `qwen3.5-omni-flash-realtime-2026-03-15` (□□□□□□□□□□ in Thai, `deploy` in Latin); one rendered `deploy` as □□□□□; one kept `framework` in Latin. Same token counts (70 audio, 97 text, 29-33 output), about 10 s wall clock, but fewer and coarser partials (9-11 against 31). Four runs is a sample, not a measurement.

Price, from the vendor's model-pricing page (Singapore, USD per 1M tokens, read 2026-11-12): input text $0.23, input audio $0.93, output text $0.70, output audio $1.87. We ask for text-only output, so the text rate applies: about $0.00011 for this clip. The page lists `qwen3.5-omni-flash-realtime` at $0.55 / $4.50 / $3.30 / $17.70 on the same four columns. `QWEN_OMNI_PRICING` used to charge the output *audio* column ($17.70, and $62 for plus) for text output; it now charges the text column ($3.30 and $12.40), so the 3.5 clip cost is now about $0.00047 rather than $0.00094 (4.4x the 3.8 price, not 8.5x).

qwen-audio-3.1-asr-flash-message and -streaming (2026-11-12)

A third Qwen wire protocol: DashScope's duplex `run-task` (a `run-task` text frame, raw PCM as binary frames, `finish-task`), against `wss://<WorkspaceId>.ap-southeast-1.maas.aliyuncs.com/api-ws/v1/inference`. Audio may only be sent after `task-started`. Results come as `result-generated` with a `sentence` (`sentence_id`, `sentence_end`, `text`) and cumulative `usage` tokens.

When partials arrive (9.2 s clip, real-time pacing, last audio chunk at 9.45 s): the Omni models send no partial before the last chunk, because the adapter sends `response.create` only at the end. Both Audio-3.1 models stream while the audio is still being sent: `-streaming` 13 partials from 1.3 s, `-message` 22 partials from 0.5 s (with `intermediate_result_enabled`; its default is final-only). Every model finishes 0.2-0.7 s after the last chunk.

Partials are not the transcript. `-message` guesses the language from the first fraction of a second, so its partials are English and Chinese junk until late in the clip (`language_hints: ["th"]` did not change the partials). `-streaming` partials are garbled early and readable from about 5 s. The partials only tell the user they are being heard.

Final text, 7 runs of `-message` paced at real time: 5 gave the good transcript (Thai script, spaced, `TypeScript/Elysia/deploy/Railway/MongoDB Atlas` in Latin, matching `qwen3.8-omni-flash-realtime`'s good runs); 2 wrote phonetic Thai for every English word (□□□□□□□□□□□□, □□□□□□□□□, □□□□□□□□□□□□□□□□□□□□) with 77 output tokens instead of 54. 5 runs of `-streaming`: always no spaces between words, always Latin `project/typescript/framework/elysia/deploy/railway` in lowercase, and `MongoDB Atlas` wrong in a different way each time (`mongodbb atlas`, `mongodbbas`, `mongodbbasas`). Neither run-to-run variation is a measurement; n is small.

Faster than real time (the CLI's `--fast`, every chunk sent back to back; 3 runs each, 9.2 s clip): all three new models accept it. `qwen3.8-omni-flash-realtime` 1.07-1.15 s, `-message` 0.83-0.86 s, `-streaming` 1.98-2.58 s. The text and token counts are the same as when paced at real time (3.8 and `-message` the good transcript all 3 times, though the phonetic-Thai variant of `-message` is seen only when paced, so 3 fast runs do not rule it out; `-streaming` the same unspaced text with `mongodbbas` all 3 times), so nothing was dropped. One short clip, so a clip of several minutes sent as fast could still behave differently; not tested.

Tokens and price. `-message` 116 in / 54 out, `-streaming` 109 in / 45 out, the vendor's measured counts. Price, from the vendor's pricing page (Singapore, per 1M tokens): $0.93 input / $0.70 output for both, about $0.00015 and $0.00013 per clip.

Tuning the message model's parameters (2026-11-12)

`qwen-audio-3.1-asr-flash-message` takes these `parameters` on `run-task` (vendor's client-events page): `disfluency_removal_enabled` (its "native text polishing": drops fillers), `language_hints` (up to 4 codes; applies to finals), `vocabulary` / `vocabulary_id` (hotwords, weight 1-5, or 50 for a "super" hotword, at most 50 of those), `input.context` (up to 5 user/assistant turns, 400 characters a turn), `max_sentence_silence` (200-6000 ms, default 800), `vad_model`, `speech_noise_threshold`, `heartbeat`, `keep_dialect` (Chinese dialects) and `intermediate_result_enabled`. Not tried, for want of audio: `vad_model`, `speech_noise_threshold`, `keep_dialect`.

What was run: the 9.2 s Thai fixture (about 400 sessions in all, 6-56 runs per setting, several sessions in parallel

at real-time pacing), and two synthetic English clips made with ffmpeg's `flite` voice, one of spoken numbers and one full of "um/uh/you know". Counts below are runs of the same audio, not independent speakers.

English words in Thai, spelled in Thai script ("phonetic": □□□□□□□□□□□, □□□□□□□□) as a share of runs on the Thai clip: no parameters 15 of 40; `language_hints: ["th"]` 1 of 24; `disfluency_removal_enabled` 0 of 24; hotwords at weight 50 0 of 24 (at weight 5, 5 of 8); a Thai+Latin example turn in `context` 0 of 24 (a bare word list, 4 of 8); `th` + polishing 0 of 12, with weight-50 hotwords also 0 of 12. Setting `max_sentence_silence` at all (200, 800 or 3000) made it worse, 44 of 56; cause unknown, so just do not set it.

Spoken numbers (English clip): with no parameters the model wrote digits in 5 of 6 runs (`23 apples`, `0812345678`, `2026`). Every `language_hints` form tried (`en`, `th`, `th,en`, `en,th`) wrote words in 24 of 24 (`twenty three`). Polishing alone gave digits in 18 of 18, and a hint plus polishing gave digits in 30 of 30. So a hint on its own breaks numbers, and polishing repairs it. **Thai spoken numbers were not tested**: there was no Thai audio with numbers.

Fillers (English clip): off, every `um`, `uh`, `you know` and `like` is kept; on, all are removed (6 of 6), and `Well` and `So` stay.

The streaming model accepts it too, and benefits (4 runs each): on the Thai clip the text gains word spaces, a capital `Project` and a correct `MongoDB Atlas`, where without it the text is unspaced and `mongodbbas`, though □□□□ loses its last letter in 4 of 4; on the filler clip the fillers are removed 4 of 4.

Conclusion: polishing is the one setting that helped on every axis, so `qwen-audio` turns it on by default (`disfluencyRemoval: false` to turn it off).

Grok Voice Transcribe 2.0 (2026-11-12)

xAI's streaming speech-to-text, `wss://api.x.ai/v1/stt?model=grok-voice-transcribe-2.0`. Everything is configured in the query string; the server sends `transcript.created` (audio may only follow it), the client sends raw PCM frames, then `{"type":"audio.done"}`. Priced per second of audio, $0.20 per hour streaming ($0.10 per hour batch), per the vendor's announcement of 2026-09-18; OpenRouter's `x-ai/grok-stt-2.0` lists the batch rate, $0.000028 per second.

On the 9.2 s Thai fixture (about 50 sessions, some in parallel):

- **Text** is the best seen so far on this fixture: Thai in Thai script, every English word in Latin (`TypeScript`, `deploy`, `Railway`, `MongoDB Atlas`), with word spaces where a Latin word meets Thai. The one variable word is the framework name: `Elysia` (right) or `Elixir`, changing from run to run in every setting (`Elixir` more often; `Elysia` in 3 of 3 with `en` + `format`, too few runs to call that an effect). `filler_words` is off by default, so fillers are removed.
- **Partials** (with `interim_results=true`) start 1-2 s in and read well, because each is the running text of the chunk being heard. A chunk final (`is_final`) locks about 3-4 s of speech; an utterance final (`speech_final`) carries the whole stitched utterance. Without `interim_results` the first text comes at about 5.5 s.
- **`transcript.done.text` is empty.** Only the utterance finals carry the text, so the adapter builds its final from those (and falls back to the chunk finals plus the last interim if the audio ended without one).

- **Fast dump is no faster**: the whole 9.2 s clip sent at once still finished at about 13.5 s, so the provider is declared real-time only.
- **End-of-audio wait is erratic.** From `audio.done` to `transcript.done`: 0.3-1 s on most single sessions, but 11 s, 18 s, 40 s and 43 s on others, and one session never finished. It happened with `interim_results` on and off (1 in 4 off), and with the audio sent evenly or with an 800 ms burst, so it is the vendor, not those settings. Several sessions at once made it worse. Worth watching before relying on this as a default; the backend's own timeouts are what catch a session that never ends.
- **Text formatting** (`format=true`) needs a `language` and, per the vendor, works for Arabic, Chinese, English, French, German, Japanese, Portuguese, Russian, Spanish, Swedish and Vietnamese: **not Thai**. Measured on the English numbers clip (2 runs each): no language, `th`, and `th` + `format` all spell the numbers out (`twenty three`, `zero eight one two ...`); `en` + `format` writes `23`, `081-234-5678` and `2026`, but also mangled the price (`four hundred fifty BAHT` became `400 50bht`). On the Thai fixture `en` + `format` left the text intact in 3 of 3 runs. **Thai spoken numbers under an English bias were not tested**: no Thai audio with numbers.

Scribe Realtime v2 (2026-11-14)

ElevenLabs' `scribe_v2_realtime`, `wss://api.elevenlabs.io/v1/speech-to-text/realtime`, header `xi-api-key`. $0.22 per hour (the same as `scribe_v3_live`), per the vendor's pricing page. The server speaks first (`session_started`); audio is JSON (`input_audio_chunk`, base64, `commit`, `sample_rate`); the turn ends with an empty chunk and `commit: true`.

On the 9.2 s Thai fixture (about 30 sessions):

- **Text** is the same in every run: Thai in Thai script, `Project`, `TypeScript`, `framework`, `Elysia`, `deploy`, `Railway`, `MongoDB Atlas` in Latin (5 of 5 at live pace, 5 of 5 fast, 5 of 5 with `language_code=th`). `Elysia` right in all of them.
- **Partials**: none in the first 2 s (the vendor says processing starts after the first 2 s of audio), the first at about 2.2 s after audio starts, then one a second. Each *replaces* the one before. They are shaky: the first two are guesses (`Project` □□□□□□□□□□□□□□□?) and one went to `Project` □□□□`-- project` before recovering.
- **Commit to final**: 0.25-0.31 s at live pace in 5 of 5 sessions.
- **Faster than real time works, and gives the same text.** The 9.2 s clip sent all at once finished in 1.5-1.8 s (5 sessions); a 28.7 s clip in 2.5 s; a 37.3 s clip of different speech in 2.7 s. The one difference seen is the Thai spelling of `framework` in the 28 s clip (□□□□□□□□□□ fast, `framework` live).
- **A commit with under 0.3 s of uncommitted audio is refused**: `commit_throttled`, then the socket closes. A 0.1 s clip hits it; the provider pads with silence.
- **Commits sent in quick succession are throttled too**: a commit every 9.2 s of audio, with the audio sent all at once, got `commit_throttled`.
- **The vendor commits by itself** after about 36 s of audio. Sent faster than real time, that segment can arrive *after* our own commit was sent, and nothing on the wire says which committed segment answers our commit, so for a recording over 30 s the provider waits for segments to stop arriving (1 s). Checked live on the 37 s clip.

- **A clip made of the same sentence four times** lost most of its text, at live pace and fast (`Project [illegible] [illegible] | [illegible] [illegible]`); a 37 s clip of different speech came through whole. So it looks like handling of repeated audio, not the 36 s limit, but the cause is not known.
- `no_verbatim=true` removed `um/uh` (2 of 2, synthetic English), and wrote the phone number as `0812345678` and the year as `2026` in 1 of 2 runs (the other spelled them out). Default is verbatim: fillers kept, numbers as words, `BHT` for baht.
- `scribe_v3_live` (3 runs): [illegible] in Thai script, but `Framework` and `Deploy` capitalized mid-sentence, and 26 partials instead of 9. Not declared.
- Errors are `{"message_type":"auth_error","error":"[illegible]"}` then a close (bad key).

Not covered: noisy audio, real Thai speech with numbers, a session longer than a minute at live pace.

iApp realtime (2026-11-14)

iApp's `realtime-v2`, `wss://api.iapp.co.th/v3/store/speech/speech-to-text/realtime-v2`, header `apikey` (a bad or missing key is an HTTP 401 before the upgrade). The model cannot be chosen; the server's `ready` message reports `chinda-asr-realtime-v1`. Free during the vendor's launch (their docs). The client sends a JSON settings message (`language_code: "th"` is the only language, `sample_rate`, `encoding`, `latency_ms`, `end_silence_ms`); the server answers `ready`; audio is binary PCM16; the text message `end` finishes the turn, followed by the remaining finals and `done`.

On the 9.2 s Thai fixture (about 12 sessions):

- **Text**: Thai, with English words written **in Thai script** ([illegible], [illegible] for Elysia, [illegible], [illegible] [illegible]), as the vendor's docs say; `deploy to Vercel` came out as [illegible]. The same text in the live run and the fast runs at the default settings.
- **English speech is not supported**: the synthetic English clips came back as Thai nonsense ([illegible] [illegible]). A 37 s mixed clip came through fine in its Thai parts and mangled in the English ones, with a few Latin words slipping through.
- **Partials**: 14 at the default `latency_ms` (560), the first at 1.1–1.5 s, about every 0.6 s, each replacing the one before. At `latency_ms` 80 there are 39 (the first at 0.8 s), at 320 24, at 1120 7–14 (the first at 1.2 s).
- **`latency_ms` changes the text**: 80 wrote [illegible] (dropping the [illegible] of MongoDB) in 2 of 2 runs, and 320 [illegible]/[illegible]; 560 wrote [illegible]. So the provider keeps the vendor's default.
- **End to done**: 19–26 ms at live pace; the sentence's final arrives about 0.45 s before the audio ends (the clip ends with silence longer than `end_silence_ms`).
- **Faster than real time works, and gives the same text**: the 9.2 s clip all at once finished in 1.7 s (0.75 s in the provider), a 37 s clip in 2.5 s.
- `TOO_FAST` (`more than 120 s of audio is waiting; send audio in real time`) ends the session when too much audio is waiting: a 150 s clip sent at once got it 0.5 s in, with no final text. The provider therefore sends up to 90 s at once and the rest at twice real time: 149 s took 30 s and 298 s took 106 s, no error, so the vendor clears that backlog at least twice as fast as it arrives.

- **Concurrency**: two sessions at once worked; a third on the same key got HTTP 429.
- A 0.1 s clip finishes normally with no text.

Not covered: noisy audio, real Thai speech with numbers, `end_silence_ms` variations, the BUSY limit (20 live sessions at launch, per the docs).

Pacing: fast dump vs realtime

`qwen-omni/qwen3.5-omni-flash-realtime-2026-03-15`, same fixture, 20 runs per condition through the `vxasr` CLI (`--fast` sends frames with no delay; realtime sends 3200 bytes each 100 ms).

Every run in both conditions returned a transcript. No hangs, no errors.

condition	wall clock	runs
`--fast`	0–1 s	20
realtime	9–10 s	20

The transcripts differ only in how the framework name is rendered:

rendering	`--fast`	realtime
`Elysia`	13	17
`Alecia`	4	3
`Alembia`	2	0
`Aleph.js`	1	0

65% vs 85% `Elysia`. **Fisher's exact (two-tailed): p = 0.273 — this sample does not establish a pacing effect.** The direction was consistent across two independent samples (2/5 vs 4/5, then 11/15 vs 13/15), and `--fast` produced two renderings realtime never did, but neither observation is significant at this n. Distinguishing 65% from 85% needs roughly 70 runs per condition.

Contrast BytePlus's bi-directional `bigmodel` mode, which **hangs outright** under a fast dump (see its section) — but see below: that is not the mode this app ships.

Cost is unaffected by pacing: this model bills per token, not per second.

BytePlus `bigmodel_nostream` (the shipped mode), fast dump vs realtime

The "BytePlus hangs on a fast dump" finding above is about the bi-directional `bigmodel` mode, which the catalogue deliberately does not declare (no `language` support — see its section). The mode actually shipped, `bigmodel_nostream`, had never been fast-dump tested. Run 2026-07-26, same fixture, through the `vxasr` CLI:

condition	wall clock	outcome
`--fast`, `language` omitted, n=3	1.4–2.2 s	all 3 completed; no hang. Transcripts match the realtime row's shape (English/Chinese nonsense)
`--fast`, `BYTEPLUS_LANGUAGE=th-T`	1.7 s	correct Thai, byte-for-byte the same rendering as the realtime `th-TH` row above
realtime (100 ms/chunk)	~9.2 s+	(existing row above)

`bigmodel_nostream` does not hang on a fast dump. Billing unaffected (still `byteplus:seedasr:seconds` quantity 10). This mode is also documented to withhold partials until 15 s of audio or the final packet regardless of pacing, so a fast dump costs it nothing it wasn't already giving up.

Also reconfirmed on this run: `qwen/qwen3-asr-flash-realtime-2025-10-27` (raw and `+groq`) and `qwen-omni/qwen3.5-omni-flash-realtime-2026-03-15` all complete a `--fast` dump in under 1.5 s with no hangs, consistent with the existing rows above.

Net: every configuration currently in `builtinConfigurations` (Qwen ASR, Qwen Omni, BytePlus `bigmodel_nostream`, each raw or `+groq`) tolerates a fast dump. The only "hangs on fast dump" case ever observed is the BytePlus mode the app does not use. The eval fan-out's fixed 100 ms/frame pacing (`apps/website/src/evalRun.ts`) is accordingly a real speed-up opportunity — its comment's justification ("BytePlus hangs outright on a fast dump") turns out to describe an undeployed mode, not the shipped one.

`gummy-realtime-v1`

Surfaced only as a value for `input_audio_transcription.model`; it is not in the vendor's speech-to-text model list. Addressed **directly** over the native run-task protocol with the same key:

model	result
`gummy-realtime-v1`	**error** — `AccessDenied: Access denied.` (with and without `language_hints`)

Alibaba Cloud DashScope — Fun-ASR

Native run-task protocol. `wss://dashscope-intl.aliyuncs.com/api-ws/v1/inference`, `Authorization: Bearer`, JSON `run-task` → wait `task-started` → **raw binary** audio frames → `finish-task`.

model	language_hints	output
fun-asr-realtime	*(omitted)*	Project Nickel repository type script, type framework shoe elisia deploy material way, lettuce mongo db atlas, penpui balikantan common.
fun-asr-realtime	["th"]	byte-identical to omitted
fun-asr-realtime	["xx"] (bogus)	byte-identical to omitted
fun-asr-realtime	["zh"]	Project Nickelodeon Pasar Typescript Child Framework Shoes Elisia Deploy Material Way LagosMongoDB AtlasPenpah Berikantan common
fun-asr-realtime-2026-02-28	*(omitted)*	byte-identical to fun-asr-realtime
fun-asr-realtime-2026-02-28	["th"]	byte-identical to omitted
fun-asr-realtime-2025-11-07	—	**error** — task-failed, error_code: ModelNotFound

zh changes the output; th and a bogus code do not. Vendor-reported billed duration for the 9.218 s clip: **9 s** (ceil(bytes / 32000) gives 10).

BytePlus — Seed-ASR

Binary-framed protocol. wss://voice.ap-southeast-1.bytepluses.com/api/v3/sauc/<mode>, X-Api-Key, X-Api-Resource-Id.

mode	resource id	audio.langua	output
bigmodel	volc.seedasr.sauc.duration (2.0)	*(unsupported on this mode)*	project Niagara typescript Chai framework Chai do deploy material way □□ Chai MongoDB Atlassian common□
bigmodel	volc.bigasr.sauc.duration (1.0)	*(unsupported on this mode)*	byte-identical to the 2.0 row
bigmodel + Groq	2.0	—	Project Niagara TypeScript. Chai framework. Chai deploys material way from Chai MongoDB Atlassian common.
bigmodel	2.0	—	**parse error** — Unexpected token 's', "sult":{"ad"□; frames appear to omit the sequence field our parser expects at byte 12
bigmodel	2.0	*(omitted)*	run 1: Project Nick and Repasa TypeScript, Chai framework to Elisia to deploy material way, Let's say Mongodb at last, and two high baricantum common.run 2: Project Nick and Repasa TypeScript Chai framework to Elisia to deploy material way Let's I'm going to be at last and put high□□□□ common
bigmodel	2.0	th-TH	□□□□□

model	output	cost
nvidia/nemotron-3.5-asr-streami 6b	— every character space-separated, unusable as-is	$0.000031
mistralai/voxtral-small	Project n□y □□□c vi□t b□ng ng□n ng□ TypeScript, s□ d□ng framework t□n l□ Elysia □□ deploy v□ Railway v□ s□ d□ng MongoDB Atlas l□ ng□□i cung c□p c□ s□ d□ li□u. — Vietnamese, not Thai	$0.000461
mistralai/voxtral-mini-	Project n□y vi□t b□ng ng□n ng□ TypeScript, s□ d□ng framework t□n l□ Elecia □□□c tri□n khai tr□n Railway v□ s□ d□ng MongoDB Atlas l□m c□ s□ d□ li□u. — Vietnamese, not Thai	$0.000154
qwen/qwen3-asr-1.7b	□□□□□□tn□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□□□□□□

`microsoft/mai-transcribe-1.5` is the one added as a `vxasr` configuration (`openrouter/microsoft/mai-transcribe-1.5`) — see `../packages/vxasr/src/providers/openrouter.ts`. Cost is what the vendor's `usage.cost` field reports directly, in USD, not a rate needing multiplication like every other section in this file.

`microsoft/mai-transcribe-2` costs about a third of `1.5`, but made two small errors that `1.5` did not (dtinth/vxbeamer#86). Not yet added as a configuration — the drop in accuracy needs a decision first.

`meta/muse-voice-transcribe-1.0` is the cleanest transcript of every OpenRouter model tried so far on this fixture, and costs under half of `mai-transcribe-1.5`. Added as the OpenRouter provider's default (dtinth/vxbeamer#86).

Meta (direct realtime API)

Run **2026-09-18**. Same model as the OpenRouter entry above (`muse-voice-transcribe-1.0`), spoken to directly over `wss://api.meta.ai/v1/asr/realtime` instead of through OpenRouter's batch wrapper — see `../packages/vxasr/src/providers/meta-muse.ts`. `PUSH_TO_TALK` mode, `PCM_16KHZ`, no `languageBias`.

output	usage	wall	first partial
□□□□□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□□□□□□□ Alessia □□□ deploy □□□□□ Railway □□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□□□□□□□□□□ — two misses the OpenRouter batch route on the same model did not make: □□□□□□□□□□ (transliterated) instead of `framework` in Latin, and `Elysia` came out as `Alessia`	$0.0005 (10 s billed, rounded up from the clip's 9.218 s — coarser than OpenRouter's fractional-cost billing)	5.1 s	2.4 s

Genuine streaming partials, unlike every OpenRouter model above (batch-only, no partials at all) — 24 partial updates arrived before the final. But the transcript itself is a regression from the batch route on this one clip: same underlying model, worse output. Not clear yet whether that is `PUSH_TO_TALK` mode specifically, missing `languageBias`, or just this endpoint's own defaults differing from whatever OpenRouter's wrapper sends — one run is not a verdict. Added as a configuration anyway (`meta/muse-voice-transcribe-1.0`, raw only) since the realtime partials are the whole reason this provider exists; the OpenRouter route stays available as a separate configuration for whichever a future eval favors.

Paxa Labs

Run **2026-09-22** (dtinth/vxbeamer#86). `POST https://api.paxalabs.com/v1/stt`, `Authorization: Bearer`, JSON body with the audio base64-encoded — a batch call, no pacing, no partials. Model `paxa-stt-lite-v1-preview`. Four clips, both `convention` values, 3 runs each.

clip	output
English, 2.0 s	But it only happens sometimes.
Thai, 2.1 s	□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
eval-set, 13.0 s	□□□□□□□□□□□ issue □□□□□□□□□□□□□□□□ sandbox provider □□□□□□□□□□□□□□□□□□□□□□□□□□□□ □□□□□□□□□□□□□□□
test-audio.bin, 9.2 s	□□□□□□□□□□□□□□□□□□□□□□□□□□□□ TypeScript □□□□□□□□□□□□□□□□□□ Elysia □□□ deploy □□□□□□ Railways □□□□□□□ MongoDB Atlas □□□□□□□□□□□□□□□□□□□□□□□□□□□□

Byte-identical across all three runs of every clip — 1 distinct output out of 3, everywhere. For comparison, `qwen3.5-omni-flash-realtime` gave 5–10 distinct out of 10 on the same clips.

It is also the only model compared in this file that needed no instruction to avoid the defects the others have: no filler tags, no spaces between Thai words, no English translated into Thai, and `Elysia` correct every run. Two small misses on `test-audio.bin`, both shared with `mai-transcribe-2`: □□□□□□□□□ for □□□□□□□□□, and `Railways` for `Railway`.

`convention=spoken` and `convention=written` produced **identical** output on all four clips. Per the vendor they differ on numbers (`1,250`□□□ vs □□□□□□□□□□□□□□□□□□□□□) and the repetition mark □, and none of these clips contains a number, a unit, or a repeated word — so this is a gap in the fixtures, not a finding about the modes.

Cost: 8.33 credits per minute, 10,000 credits for ฿329 → **฿16.45/hour** (~$0.47 at ฿35/USD).

Realtime WebSocket

Run **2026-09-24** (dtinth/vxbeamer#86). `wss://api.paxalabs.com/v1/stt/live`, `Authorization: Bearer` on the upgrade request, model `paxa-stt-lite-realtime-v1-preview`. A `start` JSON frame, raw binary PCM, then `{"type":"end"}`. See `../packages/vxasr/src/providers/paxa-realtime.ts`.

test	result
`end` → final, realtime pacing, n=6	**52–58 ms**. The vendor detects the end of speech, so the final often arrives before the audio ends.
recording cut at 5 s, mid-speech	final 327 ms after `end`, with the text up to the cut
fast dump	accepted: 0.7 s for the 9.2 s clip, 1.0 s for a 21.4 s clip
audio before `started`, 100 ms frames	both accepted (the docs show 20 ms frames)
wrong key	HTTP 401 on the upgrade
3 s of silence	`done` with `turns: 0`
partials	about one each 3 s

A pause starts a new turn, and each turn gets its own final `transcript` frame: a 3 s pause between two copies of the clip gave `turn: 1` and `turn: 2`. The adapter joins the turns into one final.

Transcript as the batch endpoint, except for `Railway`/`Railways`: with realtime pacing, 6 of 8 finals had `Railway`; batch and fast dump had `Railways` every time. Too few runs to call it an effect of pacing.

Charged at **12.5 credits/min** against 8.33 for batch, for the whole connection, silence included: the 21.4 s two-turn run was charged 21.4 s. **฿24.68/hour** (~$0.71 at ฿35/USD).

Batch providers, timing compared

Run **2026-09-22** (dtinth/vxbeamer#86). All four are batch HTTP endpoints, so this is like for like. Interleaved — each iteration calls all four back to back, so network drift lands on every provider equally rather than on whoever happened to run during it. 8 runs each, wall time of the whole call.

provider	2 s clip (median)	13 s clip (median)	cost/hour
`paxa-stt-lite-v1-preview`	**568 ms**	**900 ms**	$0.470
`microsoft/mai-transcribe-2`	850 ms	952 ms	**$0.109**
`microsoft/mai-transcribe-1.5`	1407 ms	1682 ms	$0.391
`meta/muse-voice-transcribe-1.0`	1834 ms	3856 ms	$0.181

Timing is from this machine, whose route and bandwidth are its own; the ordering is the finding, not the absolute figures. Costs are normalised from the per-request figures recorded above against the 9.218 s fixture.

`muse-voice-transcribe-1.0` was the OpenRouter default until this run. It had been chosen on transcript quality on one fixture and never timed: it is four times slower than the other two, and produced the worst transcript of the three here (□□□□□ for `issue`, and Thai transliteration where the others keep `sandbox provider` in Latin). `mai-transcribe-2` replaced it as the default — fastest, cheapest, and its 13 s transcript was identical to Paxa's.

Latency by clip length, Paxa alone, 10 runs each: 0.5 s → 384 ms, 2 s → 433 ms, 5 s → 568 ms, 13 s → 622 ms. Fitting those gives **~408 ms fixed plus ~18 ms per second of audio** — overhead-dominated, so 26× the audio costs 1.6× the time. (Measured in isolation; the interleaved figures above are higher, which is why they are the ones quoted for comparison.)

Google Gemini 3.5 Transcribe

Run **2026-09-26** (dtinth/vxbeamer#86). Tried three ways. Clips: `test-audio.bin`, the 2.0 s English and 2.1 s Thai clips, and the 13.0 s eval-set clip from the Paxa section.

route	time to final	partials	cost/ hour
OpenRouter `google/gemini-3.5-transcribe`	4–6 s per request, any clip length (22 runs)	none	$0.18
Gemini API batch (`v1beta/interactions`)	5.2–5.8 s	none	$0.30
Gemini API Live, `gemini-3.5-transcribe-live`	0.3–0.7 s after the audio ends	about each 0.5 s	$0.54

OpenRouter's reported cost is the input tokens only (25 audio tokens per second at $2/1M); the Gemini API prices are from Google's pricing page. See `../packages/vxasr/src/providers/gemini-live.ts` for the Live protocol.

`test-audio.bin` was perfect on every run of all three routes, and so were the English and Thai clips. The batch model gave byte-identical output across 5 runs of each clip. On the eval-set clip the batch model wrote □□□□□ for `issue`; `custom_vocabulary: ["issue"]` fixed it.

Live, with the vendor's voice detection (the default). Each pause ends a turn with its own final. After `audioStreamEnd`, a turn in progress gets its final, but if nobody is speaking the service sends nothing more. A whole clip sent at once never produced a final.

Live, push-to-talk (`automaticActivityDetection: {disabled: true}`, `activityStart`/`activityEnd`). One final per recording, and `ACTIVITY_END` always follows `activityEnd`, even for silence. A whole clip sent at once works: 3.3 s for the 9.2 s clip.

Push-to-talk dropped sentences on clips joined from separate recordings: [A] the `test-audio.bin` sentence, [B] the Thai clip, [C] the English clip. The same result on every run (3 of 3):

clip	push-to-talk	push-to-talk + `SMART`	voice detection + `SMART`
A, 3 s pause, B, 2 s pause, C	complete	C dropped, B misheard	complete
A, 0.3 s pause, B	B misheard	B dropped	complete
eval-set clip	complete	complete	complete

"Misheard" is □□□□□□□□□ for □□□□□□□□. Push-to-talk + `SMART` + `customVocabulary: ["issue"]` also dropped the second half of the eval-set clip (4 of 4). Push-to-talk + `SMART` is what the `gemini` configuration uses, chosen with this known (dtinth/vxbeamer#86).

Microsoft MAI-Transcribe-2 Streaming (Vercel AI Gateway)

Run **2026-10-02** (dtinth/vxbeamer#86). `microsoft/mai-transcribe-2-streaming`, offered only through the Vercel AI Gateway, reached with the AI SDK's `experimental_streamTranscribe` (`ai` 7.0.127, `@ai-sdk/gateway` 4.0.103), PCM 16 kHz. See `../packages/vxasr/src/providers/ai-gateway.ts`.

clip	final after the audio ends	transcript
`test-audio.bin`, realtime, n=5	342–491 ms	as batch `mai-transcribe-2`: correct except `Railways`; `framework` in Latin or Thai varies
`test-audio.bin`, fast dump	3.2 s	the same
A, 3 s pause, B, 2 s pause, C (Gemini section)	348 ms	**all three sentences**
A, 0.3 s pause, B	433 ms	both sentences, □□□□□□□□ correct
the same sentence twice, 3 s pause	349 ms	both copies
eval-set clip, 13.0 s	352 ms	□□□□□□□□□□□□ `issue` □□□□□□... (fillers kept, `issue` correct)
English 2.0 s / Thai 2.1 s	345 / 329 ms	both correct
3 s of silence	344 ms	empty final

Partials come about every 0.3 s (31 on the 9.2 s clip). The stream gives `transcript-delta` (finalized text) and `transcript-partial` (the provisional text *after* the latest delta), then one `transcript-final` with the whole transcript once the input ends.

For silence, the SDK's `text` promise rejects with "No transcript generated" even though `fullStream` ended cleanly with an empty final.

After the stream finishes, the SDK closes the WebSocket with code 1000, but its TLS socket stays open for about 10 s more before it closes. A short CLI run therefore exits about 10 s after printing its transcript.

$0.54 per hour of audio (the gateway's listed price).

Typhoon (SCB 10X)

Run **2026-09-12**. Same OpenAI-compatible transcription shape as OpenRouter: `POST https://api.opentyphoon.ai/v1/audio/transcriptions`, `Authorization: Bearer`, `multipart/form-data` with `model` and `file`. One request, no pacing, no partials (dtinth/vxbeamer#86).

model	output	usage	wall
`typhoo`	□□□□□□□□□□□□□□□□□□□□□□□□□□□□ `Thaiscribe` □□□□□□□□□□□□□□□□□□□□ □□□□□□□□ `Deply` □□□□□ `Realway` □□□□□□ `mango Debe Adlus` □□□□□□□□□□□□□□□□□□□□□□□□□□□□ — every English loanword garbled: `TypeScript`→`Thaiscribe`, `Elysia`→□□□□□□□□, `deploy`→`Deply`, `Railway`→`Realway`, `MongoDB Atlas`→`mango Debe Adlus`	460 input (audio) + 74 output = 534 tokens, no cost figure in the response	1.6 s

The response carries no dollar cost, only a token count (`usage.type: "tokens"`) — the docs don't publish per-token pricing either. Not added as a configuration: on this fixture every English technical term came out wrong, worse than every other Thai-capable model already compared above.

Repeat runs

Identical input, repeated. **These are the same request each time.**

`qwen3-asr-flash-realtime`, pacing compared, n=3 each

pacing	wall	outputs (first word)
fast dump (no delay)	0.65 s	□□□□□□□□ ×3
realtime (100 ms/chunk)	9.49 s	`project` ×2, □□□□□□□□ ×1

Partial counts: 56 (fast dump) vs 84 (realtime). Usage quantity **10** in both — billing did not change with wall time.

Fisher's exact on 3/3 vs 1/3 ≈ **0.4**; this sample does not establish a pacing effect.

`qwen3-asr-flash-realtime` + Groq, realtime, n=3

run	output (first word)
1	`project`
2	`project`
3	□□□□□□□□

Same audio, same configuration, same pacing.

`qwen3.5-omni-flash-realtime-2026-03-15`, prompt compared, n=10 each

Run **2026-09-20** (dtinth/vxbeamer#86). Three clips, four instructions, ten runs each. Two defects counted separately: `<fil>` filler tags with slash-separated alternate readings (`Issue/□□□□`), and spaces inserted between Thai words, which Thai does not use.

Clip A is a 13.0 s eval-set recording that reproduces the tags; clip B is a 2.1 s Thai clip that reproduces the spacing; clip C is `test-audio.bin`, kept as a control because its English technical terms *need* their surrounding spaces.

instruction	A: tags	A: alternates	B: spaces	C: Latin terms
`Transcribe the user's audio verbatim` (previous)	10/10	10/10	3 per run, 10/10	preserved
previous + Thai/Latin script rule	10/10	10/10	0/10	preserved
previous + script rule + "never output tags such as `<fil>`"	10/10	10/10	—	preserved
`Write out exactly the words the speaker says` (current)	**0/10**	**0/10**	**0/10**	preserved

Forbidding the behaviour did not work. Adding "never output tags in angle brackets such as `<fil>`" to the old wording left it at 10/10. Removing the word *transcribe* did, taking both defects to 0/10 — the working theory being that "transcribe … verbatim" selects a transcription-annotation register where those tags belong, and naming the token without leaving that register may prime it.

Incidental, on clip C: the previous wording returned *Alecia* / *Alesia* / *Elysia* across runs, the current wording *Elysia* in all ten. One fixture and not what the instruction was aimed at, so noted rather than claimed.

A first version of the new wording translated English into Thai. It stated the Thai script rule unconditionally, which reads as a claim about the output rather than a rule about Thai. Re-run on a 2.0 s English clip, n=10:

instruction	output in Thai script	transcript
`Transcribe  verbatim` (older)	0/10	`But it only happens sometimes.`
unconditional Thai rule	**10/10**	□□□□□□□□□□□□□□□□□□□□□□□□□□□□ and four other renderings
current, language named + Thai rule conditional	0/10	`But it only happens sometimes.`

The current wording holds every result above on the Thai clips while fixing this, so the language clause costs nothing. A prompt that describes the expected output in one language will produce that language.

Counting caveat: "spaces between Thai words" counts any space with Thai on both sides, but Thai does use spaces at clause boundaries. On clip A the two remaining spaces sit around the fillers □□□ and □□□, which is correct typography rather than the defect.

Errors and timeouts observed

what	result
BytePlus `bigmodel`, audio dumped as fast as the socket accepted	**60 s timeout, no final** — vendor docs specify a 100–200 ms send interval
`qwen3-omni-flash-realtime` on the Qwen realtime protocol	60 s timeout, no result
`fun-asr-realtime-2026-02-28`	`task-failed` / `ModelNotFound` — the vendor's model list documents this id
DashScope intl key against Beijing (`wss://dashscope.aliyuncs.com`)	**401 at the WS handshake** — keys are region-scoped

Vendor-documented language support, for comparison

From the Model Studio model list. Recorded because some rows above differ from it.

model	documented languages
`qwen3-asr-flash-realtime` (all snapshots)	incl. **Thai**
`fun-asr-realtime`, `fun-asr-realtime-2025-11-07`	incl. **Thai**
`fun-asr-realtime-2026-02-28`	Chinese, English, Japanese
`fun-asr-realtime-2025-09-15`	Chinese, English
`paraformer-realtime-v2`	zh, en, ja, yue, ko, de, fr, ru
BytePlus, `language` omitted	Mandarin, English, Cantonese, Shanghainese, Minnan, Sichuan, Shaanxi

Reproducing

```
vp run vxasr#build
node --env-file=.env packages/vxasr/dist/cli.mjs <configuration-id>
↪testdata/test-audio.bin
```

`vxasr --list` shows the configuration ids.

Rows the shipped CLI **can** reproduce: the pinned Qwen snapshots (raw and `+groq`), `byteplus/bigmodel_nostream` with `BYTEPLUS_LANGUAGE=th-TH`, and the Qwen Omni **realtime** rows under `qwen-omni/` — the undated ids in the table below are now pinned to `qwen3.5-omni-flash-realtime-2026-03-15`, `qwen3.5-omni-plus-realtime-2026-03-15`, and `qwen3-omni-flash-realtime-2025-12-01`.

Rows it **cannot**: the floating `qwen3-asr-flash-realtime` id and `byteplus/bigmodel` (both removed from the catalogue), the Qwen Omni **offline** rows and the transcription-sub-service rows (no adapter — the sub-service is deliberately silenced), and everything under Fun-ASR. Those came from throwaway scripts against the same endpoints.

2.8 Backend

The backend (`apps/backend`) is a Node.js server built with Hono. It receives PCM audio over a WebSocket, runs it through one or more ASR configurations with *vxasr* (page 16), streams the transcripts to clients over SSE, and exposes the REST API described in *API* (page 11).

It is distributed as a Docker image, `ghcr.io/dtinth/vxbeamer`; see *Getting started* (page 4) for a compose file and *Configuration* (page 8) for the environment variables.

Development

From the repository root:

```
vp install
vp run dev:backend
```

The server needs `DASHSCOPE_API_KEY` for the default provider (or `ASR_PROVIDER=mock`, which needs no key) and, to sign clients in, an OIDC provider or `API_KEYS`; see *Configuration* (page 8). Note that `OIDC_SECRET` falls back to `local-dev-secret`, which is only fit for local development. Run the *website* (page 39) alongside it and pair it with the backend, as described in *Pairing a client* (page 7).

2.9 Website

The website (`apps/website`) is the React PWA: the frontend you add to your home screen. It is deployed statically; the hosted instance at vxbeamer.vercel.app[11] connects to whichever backend you pair it with.

See *Usage* (page 5) for how to record, copy, delete and beam transcripts.

Keyboard shortcuts

Keyboard shortcuts (ignored while typing in a text field): `c` copies the latest finished transcript, `r` toggles recording, and holding Space is push-to-talk — it starts recording on press, and on release stops and copies the transcript automatically once it finishes.

Development

From the repository root:

```
vp install
vp run dev:frontend
```

2.10 vxbeamer Desktop

A Tauri-based desktop wrapper for the vxbeamer web app. It serves the built website assets inside a native webview and adds desktop-specific features such as acting on swipe events received from another device (copy to clipboard or paste into the active application).

[11] https://vxbeamer.vercel.app

Prerequisites

Requirement	Notes
Node.js ≥ 20	Managed by `voidzero-dev/setup-vp` in CI; use nvm[12] or fnm[13] locally
Vite+ (vp)	Install and expose the global `vp` command as described by the Vite+ toolchain
Rust (stable)	Install from https://rustup.rs/
Platform system libraries	See table below

Linux system libraries

```
sudo apt-get update
sudo apt-get install -y \
  pkg-config \
  libglib2.0-dev \
  libgtk-3-dev \
  libwebkit2gtk-4.1-dev \
  libayatana-appindicator3-dev \
  librsvg2-dev \
  libxdo-dev \
  libssl-dev
```

macOS

No extra system libraries are required. Xcode Command Line Tools must be installed (`xcode-select --install`).

Windows

No extra system libraries are required. Make sure the Microsoft C++ Build Tools[14] (or Visual Studio with the "Desktop development with C++" workload) are installed.

Development

Start the live-reloading development server (runs the website dev server and opens the Tauri webview against it):

```
# From the repository root
vp install

# Then either:
vp run desktop#dev
# or, from apps/desktop:
cd apps/desktop
```

(continues on next page)

[12] https://github.com/nvm-sh/nvm

[13] https://github.com/Schniz/fnm

[14] https://visualstudio.microsoft.com/visual-cpp-build-tools/

(continued from previous page)

```
vp run website#dev   # in a separate terminal
node_modules/.bin/tauri dev
```

The website dev server listens on port **20470** (see `tauri.conf.json`).

Building for production

From the **repository root**:

```
vp install
vp run desktop#build
```

Or from the `apps/desktop` directory:

```
node scripts/prepare-web-assets.mjs   # builds the website and copies dist/
node_modules/.bin/tauri build
```

`prepare-web-assets.mjs` builds the website via the global `vp run website#build` and copies the output into `apps/desktop/dist/`, which Tauri then bundles.

Output bundles

Platform	Location
Linux `.deb`	`src-tauri/target/release/bundle/deb/`
Linux `.AppImage`	`src-tauri/target/release/bundle/appimage/`
macOS `.dmg`	`src-tauri/target/release/bundle/dmg/`
Windows `.msi`	`src-tauri/target/release/bundle/msi/`
Windows `.exe` (NSIS)	`src-tauri/target/release/bundle/nsis/`

CI

The Build Desktop[15] workflow builds the app on all three platforms (Linux, macOS, Windows) on every push to `main` and on pull requests. The resulting bundles are uploaded as GitHub Actions artifacts. When a release is published from the npm publish workflow, the same build is reused to attach those bundles to the GitHub release.

2.11 vxbeamer transmitter

Voice input for vxbeamer from a phone, or from a Wear OS watch (built and tested against a Samsung Galaxy Watch 5) that never talks to the internet itself (dtinth/vxbeamer#86).

Two separate Android apps live here, in one Gradle project:

- **`wear/`** — runs on the watch. One button. Tap to start: captures the mic (16 kHz / 16-bit / mono, the exact format vxbeamer's own `/ws` already expects) and streams it to the phone over Bluetooth, using the Wear OS Data Layer's `ChannelClient`. Tap again to stop.

[15] https://github.com/dtinth/vxbeamer/blob/main/.github/workflows/desktop.yml

- **`mobile/`** — runs on the paired phone. Records from its own microphone, and receives audio relayed from the watch.

  Recording never waits for the network. Audio goes to local storage as it is spoken, and uploading is a separate, retryable thing that happens to it afterwards — the same shape as a voice memo. Tapping stop is instant whatever the connection is doing, nothing is lost when an upload fails, and a bad transcript can be transcribed again from the audio still on the device.

  In the normal case the two run at once: the uploader tails the file while the microphone is still writing it, so a transcript arrives as promptly as it would from a live socket. The same code path run later against a finished file is a retry.

  Three ways to start, because none of them is best everywhere:

	Reach	Notes
Full screen	Open the app	Record button at the bottom, in thumb reach, above the history
Floating button	One tap, anywhere	Draggable, live transcript and level meter. Uses "draw over other apps", which banking apps can suppress on Android 12+, so it is opt-in and switchable off
Quick Settings tile	Swipe down, tap	No permission, and no app can suppress it

  A second tile, "Floating button", switches the floating button on and off from the same shade, so it need not be left on all the time. With the permission not yet granted, tapping it opens the app, where the toggle asks.

  Leaving the app mid-recording drops it into a square picture-in-picture window. That window is a **status readout, not a control** — a PiP window never delivers touches to its content, so its stop button is the `RemoteAction` revealed by tapping it.

  Watch audio goes through the same store, so relayed recordings get the same history and the same retry as local ones.

  Any recording's audio can be exported as a WAV, in the same format the eval fixtures use, for listening to or testing a transcription against. Audio is capped at 10 MB in total, oldest dropped first — except audio that has not been transcribed yet, which is never deleted to stay under the cap, since that would destroy the only copy of what was said.

The phone app pairs with your backend: open `/pair` on the backend wherever you are signed in, then scan the QR code it shows (Google's code scanner, no camera permission) or paste its link into the app. The link carries the backend's address, so there is no URL to type.

Status

The **phone app works** — transcribing from its own mic is verified on real hardware (dtinth/vxbeamer#86).

The **watch relay is still unverified**: it has never completed a recording on a real watch, and the watch it was written for has a failing battery. It compiles and its one known bug is fixed (the relay shares `BackendWebSocket`

with the phone, which was rejecting its own URLs until that was found on device), but treat it as untested.

This is developed in a sandbox with no emulator (no hardware virtualization), so anything not listed as verified above has only been checked for `assembleDebug` and `lintDebug` passing.

Design

Piece	Job
`Recording`, `UploadPolicy`, `RetentionPolicy`	What a recording is, what gets uploaded next, what gets kept
`RecordingStore`	Audio files and the index on disk
`AudioCapture`	Microphone to file. Knows nothing about the network
`TranscriptionUploader`	File to `/ws`, and `/sse` back. Knows nothing about the microphone
`Recorder`	Ties those together and owns the queue
`RecorderService`	Keeps it alive in the background; hosts the floating button

The split is what makes the tests possible: everything in the first four rows is free of Android APIs, so it runs on a plain JVM.

Tests

```
cd apps/watch
./gradlew testDebugUnitTest
```

Unit tests, no emulator, seconds to run — and they run in CI on every push. They cover the parts where being wrong is quiet: URL building (a swapped `ws`/`wss` scheme shipped once and took a device and a logcat to find), the `/sse` event filtering that decides whether a transcript is *yours*, the upload and retention rules, crash recovery, and the store's behaviour across a restart.

Prerequisites

Requirement	Notes
JDK 21	e.g. `mise use -g java@21`
Android SDK	`compileSdk 34`; command-line tools are enough, Android Studio is not required to build

On an Apple Silicon Mac or another arm64 Linux machine, Google's own `aapt2` (pulled in by the Android Gradle Plugin) is x86_64-only and will fail to run. See Commit451/android-arm-build-tools[16] for a drop-in arm64 replacement, or build on an x86_64 machine — GitHub Actions' runners already are, so CI needs none of this.

[16] https://github.com/Commit451/android-arm-build-tools

Building

```
cd apps/watch
./gradlew assembleDebug
```

Produces two unsigned-but-debug-signed APKs:

- `mobile/build/outputs/apk/debug/mobile-debug.apk`
- `wear/build/outputs/apk/debug/wear-debug.apk`

Installing

Both apps are personal and unpublished — there is no Play Store listing. Install straight from the APK:

```
adb install mobile/build/outputs/apk/debug/mobile-debug.apk   # to the phone
adb -s <watch-serial> install wear/build/outputs/apk/debug/wear-debug.apk   # to the
↪watch, over adb-over-wifi or a USB dock
```

Or download the built APKs from GitHub Actions (see below) and open the file directly on-device.

Pair the phone app with your backend once before trying to record from the watch.

CI

The Build Watch Apps[17] workflow builds both APKs on every push to `main` and on pull requests touching this directory, and uploads them as GitHub Actions artifacts — open the workflow run and download `vxbeamer-watch-relay-mobile-debug` / `vxbeamer-watch-relay-wear-debug` from the bottom of the page.

2.12 Decisions

Architecture decision records: why vxbeamer is built the way it is. Each record is a short page in this folder, and new ones are listed here automatically.

A model configuration, not a provider, is the unit of selection

Status: accepted. This record was written after the fact, from the decision as it is stated in the code (`packages/vxasr/src/builtin.ts`, `apps/backend/src/asr.ts`) and the `vxasr` README.

Context

vxbeamer talks to many speech-recognition vendors. A vendor's output can also be run through a post-processing step, such as the Groq formatting pass. Evals pit candidate transcripts against each other, and the winner of an eval replaces the primary answer, so everything that can be a candidate has to be comparable with the rest on equal terms.

[17] https://github.com/dtinth/vxbeamer/blob/main/.github/workflows/watch.yml

Decision

The thing a client selects, an eval votes for and the catalogue lists is a **model configuration**: a provider, a model and its post-processing chain, written `provider/model` or `provider/model+post`. Post-processing is part of a configuration's identity, not a flag on the request.

So a model raw and the same model with Groq formatting are two configurations, declared separately, and a vote for one says something different from a vote for the other. A client names a configuration with `?configuration=` and cannot ask for a post-processing step on its own.

Every model is pinned to a dated snapshot wherever the vendor publishes one. An undated id floats, because the vendor repoints it without notice, and that would let a transcript change with nothing in the repository changing, and make a vote name a moving target. A model the vendor offers only undated is declared as such, with a comment saying so.

The `mock` provider is never paired with a real post-processing step: there is deliberately no `mock/mock+groq`, so no branch in selection can wrap it by accident.

Consequences

- The list of configurations is a catalogue in `packages/vxasr/src/builtin.ts`; a new vendor or model is a new entry, and `GET /asr/configurations` lets a client discover the set instead of hardcoding it (see *Model configurations* (page 13)).
- Ids contain `+` (and sometimes `/`), so clients must URL-encode them.
- A configuration is selectable once the environment carries every credential it needs, so the set a deployment offers follows from its keys.
- Models that already produce well-formatted output are declared raw only, since a second pass costs a vote slot and a call for nothing.