STT Library — Production User Guide

This guide is for **application developers** integrating the STT Service via the **Python SDK** or the **raw HTTP API**. It covers capabilities, how to call the API in production, resiliency patterns, performance tuning, and troubleshooting.

Audience & Scope

- You **use** the service to transcribe short audio clips quickly and reliably.
- You do **not** operate the GPUs or deploy the servers (see the separate Service Ops guide for that).

What the Library Does

- **Fast speech-to-text** with confidence-based routing:
    - **Primary** model (small) runs first for low latency.
    - If primary **errors** or looks **low-confidence**, it transparently falls back to a **secondary** model (medium) for accuracy.
- **Language handling**
    - If you **pass** `language` (e.g., `"en"`, `"hi"`), **both** models use that language (no auto-detect checks).
    - If you **omit** `language` or set it to `None`, the service **auto-detects** and may route to fallback if language probability is low.
- **Clean responses** (final text + segments + timing + routing metadata).
- **Robust client SDK**: concurrency control, retries for transient failures, optional multi-host pooling, easy telemetry.

---

1) Install & Initialize

```
pip install httpx
```

> The SDK itself is a single Python module (`stt_sdk.py`) with no heavy deps. Drop it into your codebase or install as your own package.

Create a client

```
import asyncio, pathlib
from stt_sdk import STTClient, STTClientConfig

async def main():
    cfg = STTClientConfig(
        base_url="http://PRIMARY_HOST:8081",
        api_key="primary_key_abc",      # ask your ops team
        max_concurrency=4,              # client-side parallelism
        timeout_s=60.0,                 # request timeout
    )
    client = STTClient(cfg)

    audio = pathlib.Path("sample.wav").read_bytes()
    result = await client.transcribe_bytes(audio, language="en",
beam_size=1, vad_filter=True)

    print(result.pretty_seconds())      # timings summary
    print("Text:", result.text)         # final transcription
    print("Final model:", result.final_model)  # "small" or "medium"
    print("Language:", result.language)        #
{'code':'en','prob':..., 'source':'forced'}

    await client.close()

asyncio.run(main())
```

> **Tip**: Keep `beam_size=1` and `vad_filter=True` for best latency (these are also the SDK defaults).

---

2) Basic Usage Patterns

2.1 Force a Language (recommended for short clips)

```
res = await client.transcribe_bytes(audio_bytes, language="en")
```

- Disables language-probability gating.
- Reduces unnecessary fallbacks on very short/telegraphic audio.

2.2 Auto-Detect Language

```
res = await client.transcribe_bytes(audio_bytes, language=None)
```

- Service detects language on the primary.
- Routing to fallback can occur if:
    - average logprob is below threshold,
    - language probability is below threshold,
    - the clip is too short.

2.3 Tune Routing (optional)

```
res = await client.transcribe_bytes(
    audio_bytes,
    language=None,               # autodetect
    conf_threshold=0.50,         # default 0.55 (lower = fewer
fallbacks)
```

```
    langprob_threshold=0.65,     # default 0.70 (only used when
autodetect)
    min_words_for_conf=2,        # default 3  (use 2 for short
commands)
)
```

2.4 Get Segment Detail & Timestamps

The response always includes `segments` (start/end seconds + text). Word-level timestamps are disabled by default for latency; you can request them:

```
res = await client.transcribe_bytes(audio_bytes, language="en",
word_timestamps=True)
```

> Word timestamps add overhead; use only if you need them.

---

3) Response Anatomy

`STTResult` fields:

- `text` – concatenated transcript string.
- `segments: List[dict]` – each `{"start": float, "end": float, "text": str, "avg_logprob": float|null, "words": int}`.
- `language: {"code": str|None, "prob": float|None, "source": "forced"|"auto"}`.
- `timings_ms: {"queue_wait", "primary_infer", "fallback_infer", "total"}`.
- `server` – server metadata & request id.
- `gpu` – GPU snapshot (informational).

- `queue_depth` – point-in-time queue snapshot.
- `confidence` – routing inputs & thresholds.
- `routing` – `{fallback_used, fallback_attempted, fallback_error, final_model, reason}`.
- `final_model` – `"small"` or `"medium"`.
- `fallback_used: bool`.
- `headers` – all `X-*` response headers from the server.

Helper methods:

- `result.pretty_seconds()` → human-friendly timing summary.
- `result.low_confidence()` → bool (uses server header or payload flags).

---

4) Production-Grade Error Handling

The SDK already retries **429/502/503/504** with exponential backoff. For your app logic, add a wrapper like:

```
import httpx

try:
    res = await client.transcribe_bytes(audio, language="en")
    # use res.text / res.segments ...
except httpx.HTTPStatusError as e:
    r = e.response
    # Server attaches useful diagnostic headers on failures:
    print("status:", r.status_code)
    print("X-Req-Id:", r.headers.get("X-Req-Id"))
    print("X-Error-Type:", r.headers.get("X-Error-Type"))
    print("X-Error-Message:", r.headers.get("X-Error-Message"))
```

```
    print("X-Fallback-Error:", r.headers.get("X-Fallback-Error"))
    # Optionally inspect JSON error body:
    try:
        print("body:", r.json())
    except Exception:
        pass
except Exception as e:
    # network/client-side issues
    print("fatal:", repr(e))
```

Common cases

- `401/403` → missing/invalid API key.
- `413` → payload too large (see §6).
- `429` → client rate-limited; the SDK retries, but you should also **backoff** or reduce concurrency.
- `502` from primary → primary tried fallback and *that* failed (transient); SDK retries; if still failing, inspect headers above.

---

5) Concurrency & Throughput (Client-Side)

- `STTClientConfig.max_concurrency`: limits **your** in-process parallel requests to avoid overwhelming the service.
- For higher throughput / HA across multiple primaries, use **`STTPool`**:

```
from stt_sdk import STTPool

pool = STTPool(
    base_urls=[
```

```
        "http://primary-a:8081",
        "http://primary-b:8081",
        "http://primary-c:8081",
    ],
    max_concurrency_per_host=4,
    api_key="primary_key_abc",
)

res = await pool.transcribe_bytes(audio, language="en")
```

- If one host errors, the pool automatically retries another host.

---

6) Audio Guidelines (Latency & Limits)

- The server enforces a **request size cap** (`MAX_UPLOAD_MB`, ask Ops; default 16 MB). A `413` means your multipart request exceeded it.
- Prefer **short clips** (a few seconds). For long media:
    - **Split** client-side and send in chunks (you stitch transcripts together).
    - Keep **mono, 16 kHz or 32 kHz WAV** for predictable performance (typical Whisper inputs).
- Keep `beam_size=1` unless you absolutely need a small accuracy bump.

---

7) Authentication

- Use the **Bearer token** your Ops team provides:
    - SDK: set `api_key="..."` in `STTClientConfig`.

- Raw HTTP: `Authorization: Bearer <key>` header.

Keys can rotate at any time; handle `401/403` by fetching a new token from your secret store.

---

8) Telemetry & Health From the Client Side

- The SDK can expose a telemetry hook for your own logging/metrics:

```
def on_telemetry(event):
    # event includes: ok, attempt, headers, timings_ms, routing, etc.
    print(event)

client = STTClient(STTClientConfig(base_url=..., api_key=...),
on_telemetry=on_telemetry)
```

- You can fetch a small JSON metrics snapshot (not Prometheus) from the primary:

```
stats = await client.server_metrics()   # GET /v1/metrics
print(stats)                            # queue_depth, in_flight, gpu,
server_commit...
```

---

9) Raw HTTP API (for non-Python clients)

Endpoint

```
POST /v1/transcribe
Content-Type: multipart/form-data
Authorization: Bearer <api-key>
```

Form fields

Field	Type	Default	Notes
`audio`	file	n/a	required (WAV recommended)
`language`	text	*omit*	If set (e.g., `en`, `hi`), **forces** language on both models
`beam_size`	int	`1`	`1` recommended for latency
`vad_filter`	bool	`true`	pass `true`/`false`
`word_timestamps`	bool	`false`	increases latency; use only if needed
`conf_threshold`	float	`0.55`	lower → fewer fallbacks
`langprob_threshold`	float	`0.70`	only when autodetect
`min_words_for_conf`	int	`3`	lower to avoid fallbacks for very short clips

Success (200 JSON)

Contains all fields shown in §3 (text, segments, language, timings, confidence, routing, …).

Important Headers (always on 200)

- `X-Req-Id`
- `X-Queue-Wait-ms`, `X-Primary-Infer-ms`, `X-Fallback-Infer-ms`, `X-Total-ms`
- `X-Fallback-Used`, `X-Final-Model`, `X-Conf-Below`
- `X-LangProb-Primary`, `X-LangProb-Threshold`, `X-Lang-Check-Applicable`

Error example (502 JSON)

```
{
  "error": {
    "type": "UpstreamFailure",
    "message": "Primary failed or low-confidence; fallback failed",
    "primary_error": "...",
```

```
    "fallback_error": "...",
    "request_id": "abcd1234"
  }
}
```

Headers include: `X-Error-Type`, `X-Error-Message`, `X-Fallback-Error`.

`curl` example

```
curl -sS -X POST "http://PRIMARY_HOST:8081/v1/transcribe" \
  -H "Authorization: Bearer primary_key_abc" \
  -F "audio=@sample.wav" \
  -F "language=en" \
| jq .
```

---

10) Performance Playbook (for Users)

- **Force language** (`language="en"`) for short or command-like audio.
- Keep clips short; if long, **chunk** on the client and stitch results.
- Use **client-side concurrency** (4–8) but watch for `429` responses; if you see them, backoff.
- Tune routing only if your domain demands it; defaults are sensible:
    - `conf_threshold=0.55`
    - `langprob_threshold=0.70` (autodetect only)
    - `min_words_for_conf=3` (try `2` for terse inputs)
- Don’t over-optimize timestamps unless you truly need word-level timing.

---

11) Troubleshooting (User-Side)

I sometimes see 502 errors

- The SDK retries transient 502s. If you still see failures:
    - Print headers `X-Error-Type`, `X-Error-Message`, `X-Fallback-Error` for your logs.
    - Reduce burstiness (lower your `max_concurrency` or add jitter).
    - If you're using autodetect on very short clips, either **force language** or lower `min_words_for_conf`.

I get 413 Payload Too Large

- Reduce file size or length. Send mono WAV at 16 kHz where possible.
- Split long recordings into segments and send them separately.

Latency is spiky

- Reduce your parallelism, especially if you're sending large files.
- Keep `beam_size=1` and avoid word timestamps unless required.

Language is wrong sometimes

- Force the language: `language="en"` (or your target code).

---

12) Reference: Quick Code Snippets

Use the pool for HA

```
from stt_sdk import STTPool
pool = STTPool(
```

```
    ["http://p1:8081","http://p2:8081"],
    max_concurrency_per_host=4,
    api_key="primary_key_abc",
)
res = await pool.transcribe_bytes(audio, language="en")
```

Telemetry hook for your logs/metrics

```
def log_ev(ev): print(ev)
client = STTClient(STTClientConfig(...), on_telemetry=log_ev)
```

Graceful shutdown

```
await client.close()  # close the underlying httpx.AsyncClient
```

FAQ

Which language codes are supported?
Use standard short language tags (e.g., `en`, `hi`, `es`). If you're unsure, ask Ops for the list aligned to the deployed Whisper model.

Can I stream audio?
This version expects a single file upload per request. For long audio, split & send sequentially.

What about privacy?
The SDK sends your audio as an in-memory multipart upload; it doesn't write to disk by itself. Storage/retention policies are determined by your service operator.

If you need example wrappers for **batch processing**, **chunking long files**, or a **minimal TypeScript client**, I can generate those next.