Live Streaming API (Old)

> **Use `docs/live/live_api_doc_v2.md` (v4.1) for everything new.** This old doc (v1.1, 2026-09-14) is back only so existing app code can still be checked against it. It is **not updated** any more, and parts of it no longer match the backend. What changed for the app: `docs/changelog/frontend_changelog_2026-11-25_live.md` and the later live change logs.
>
> **Still works (same address)** — the new doc has the current request / answer:

Old section	Now in `live_api_doc_v2.md`
`POST /live/streams` create	#1 Create a live (more fields: booking, shop, event, banner)
`POST /live/streams/{id}/ingest-key`	#9 Get the ingest key
`POST /live/streams/{id}/end`	#12 End a live
`PUT /live/streams/{id}/reminder`	#7 Remind me
`GET /live/streams/{id}`	#8 Read a live
`POST /live/streams/{id}/comments`	#24 Post a comment
`PUT …/comments/mute/{accountId}` · `PUT …/comments/slow-mode`	#25 Mute a person · #26 Slow mode
`POST` / `GET …/attachments`, pin, unpin, remove	#80–#84 The shelf

> **Gone — these calls fail now:**

Old	Instead
Part 4 Spaces — every `/live/spaces/…` endpoint	**Audio lives** (`mode: LIVE_AUDIO`): Part A4 and the stage endpoints in the new doc
`GET …/comments` (read the overlay)	the room: #15 Read the room + the `COMMENT` real-time events (Part D)
`POST …/attachments/{id}/tap`	`POST /sale-taps {from: LIVE, …}` (new doc #85 note, `docs/analytics/SALE_ORIGINS_PLAN.md`)
`PROCESSING` / `REPLAY_READY`, replays over HLS from the CDN	**No recording**: when a live ends, the video is gone (only the summary stays)
Real time over SSE (`GET /events`)	Centrifugo: #13 Connect to real time, #14 Open a live's channels

Live Streaming (VP Live) API Documentation

Author: Josh S. Sakweli, Backend Lead Team **Last Updated**: 2026-09-14 **Version**: v1.1

Base URL: `http://localhost:8765/api/v1` (local) — `https://dev.api.nexgate.co/api/v1` (staging)
Media: SRS ingest `stream.dev.nexgate.co` — LiveKit `wss://livekit.dev.nexgate.co` (staging)

> **v1.1** — `spaces_api_doc.md` **has been merged into this document** (Part 4); it no longer exists as a separate file. Verified end to end on staging on 2026-09-14: a real broadcast published over RTMP, watched over HLS, ended, and replayed from the CDN. Adds Part 0. Two response changes matter to clients: **ingest URLs are now routable addresses**, not `127.0.0.1`, and the **Space stage response now carries** `url` — it previously returned a token with no server to spend it on.

Short Description: VP Live is one-to-many broadcasting: a host publishes, an audience watches over HLS through the CDN. Four surfaces sit here — **Streams** (lifecycle and ingest), **Comments** (the live overlay), **Attachments** (the commerce shelf) and **Spaces** (many speakers, an HLS audience). Media goes host → SRS → CDN → viewers; the backend never carries a frame.

Hints:

- Every endpoint needs `Authorization: Bearer <token>` unless noted.
- **There is no "go live" endpoint.** A host asks for an ingest key and publishes; the stream turns `LIVE` when *SRS reports media arriving*. A client never sets state — the media server is the authority.
- `playbackUrl` is **null until media actually flows**. Handing it out earlier gives a player a URL that 404s, which reads as broken rather than as not-started-yet.
- The comment **read** endpoint is CDN-cached and returns a **bare array**, not the platform envelope. It is the one endpoint served at volume.
- Refusals are `409` with a `code` (`404` for `NOT_FOUND`).

Standard Response Format

```
{
  "success": true,
```

```
  "httpStatus": "OK",
  "message": "Stream created",
  "action_time": "2026-08-20T10:30:45",
  "data": { }
}
```

Refusals

`409 CONFLICT` (or `404` when the code is `NOT_FOUND`). For streams, attachments and comments the envelope `message` carries the **code**:

```
{
  "success": true,
  "httpStatus": "CONFLICT",
  "message": "NOT_HOST",
  "data": { "code": "NOT_HOST", "reason": "This is not your stream" }
}
```

Spaces are shaped the other way round — `message` is the prose and `data` holds only `{ "code": … }`. Read `data.code` in both cases and you are safe.

Part 0 — Which live am I building?

There are **three** live shapes and they are not variations of one screen. Two are streams and one is a Space; they differ in who may speak, and that decides everything else — the protocol the broadcaster uses, whether a token is issued, and what the audience costs you.

The three shapes

	`LIVE_VIDEO`	`AUDIO_RADIO`	**Space**
Who speaks	**one** person	**one** person	**many** — up to 10 on stage
Carries	video + audio	audio only	audio only
Entity	stream	stream	**its own** — not a stream mode

	`LIVE_VIDEO`	`AUDIO_RADIO`	**Space**
Endpoints	`/live/streams`	`/live/streams`	`/live/spaces`
Broadcaster sends	RTMP or WHIP	RTMP or WHIP	**WebRTC to LiveKit**
Audience receives	HLS	HLS	HLS
Audience on the SFU?	no	no	**no** — listeners never are
SRS app	`live`	`radio`	`space`

> Spaces **used to be** a third stream mode. They are not any more. If you find code or an older doc treating a Space as `StreamMode.SPACE`, it is stale.

Choosing

```
How many people may speak?
     |
     +-- exactly one --> it is a STREAM     /api/v1/live/streams
     |                       |
     |                       +-- with video?  mode LIVE_VIDEO
     |                       +-- audio only?  mode AUDIO_RADIO
     |
     +-- several, and they change --> it is a SPACE   /api/v1/live/spaces
```

The question is never "video or audio". It is *how many voices*. A single host talking to an audience is a stream even when there is no camera — that is what `AUDIO_RADIO` is for. The moment a listener can be invited up to speak, you need a Space, because that requires an SFU and a live audio mix.

The rule that governs both

> **Nothing is live because your app said so.** A stream reaches `LIVE` when SRS reports that a segment closed. A Space reaches `LIVE` when the mixed audio actually arrives at SRS. Opening the stage or pushing the first byte does **not** flip the state.

So the client never sets state. It creates, it publishes, and it **watches**. Between "I pressed Go Live" and `LIVE` there is a real gap — `STARTING` for a stream — and your UI must show it honestly rather

than pretending.

You think	Actually
"I pressed go live, so I am live"	you are `STARTING`; nobody can watch yet
"The stage is open, so the Space is live"	the Space is still `DRAFT` until audio reaches SRS
"I stopped, so it is over"	`ENDED`, then `PROCESSING`, then `REPLAY_READY`

Where the audience always is

Every one of the three sends its audience to **HLS**, never to the SFU:

```
one speaker      ──RTMP/WHIP──> [ SRS ] ──HLS──> audience
many speakers ──WebRTC──> [ LiveKit ] ──mixed──> [ SRS ] ──HLS──> audience
```

That is the cost model in one picture. A thousand listeners is a thousand HLS readers of the same segments — cheap, and cacheable. A thousand listeners on the SFU would be a thousand live downstreams. **This is why joining a Space as a listener returns no token**: a token is what would put them on the SFU.

Part 1 — Streams

`api/v1/live/streams`

Why there is no "go live" button

```
HOST                          BACKEND                SRS                VIEWERS
 |                               |                    |                    |
 |-- POST /live/streams -------->| DRAFT              |                    |
 |   {title, mode}               |                    |                    |
 |                               |                    |                    |
 |   [ device-check screen; host is actually ready ]  |                    |
 |-- POST /{id}/ingest-key ----->| mint key           |                    |
 |<-- {ingestKey, whipUrl, rtmpUrl, expiresAt}        |                    |
 |                               |                    |                    |
 |======== publish (WHIP or RTMP) ==================>|                    |
 |                               |<-- hooks /publish --|                    |
```

```
|                              |  state STARTING     |   (nothing to play)|
|                              |                     |                    |
|                              |<-- hooks /hls ------|  first segment closed
|                              |  state LIVE         |                    |
|                              |  announce + notify ====================> |
|                              |                     |   playbackUrl now set
|                              |                     |                    |
|-- POST /{id}/end ----------->| ENDED               |                    |
|                              |<-- hooks /dvr ------|  recording ready   |
|                              |  PROCESSING -> REPLAY_READY              |
```

The point to make in a demo: `STARTING` exists so that announcing a stream never sends followers to an empty player. RTMP takes ~8 seconds to produce a first segment, and WHIP can take a minute. `LIVE` is the first moment the word is true for anybody except the host.

Stream state machine

```
DRAFT ──────┐
            ├──> STARTING ──> LIVE ──> ENDED ──> PROCESSING ──> REPLAY_READY
SCHEDULED ──┘                                                └──> FAILED
    └──(abandoned ~1 day, silently)──> ENDED
```

State	Meaning
`DRAFT`	Being set up. Nothing is live and nothing costs anything
`SCHEDULED`	Booked for later. Announced on entry and again when it opens
`STARTING`	Publisher reached SRS, but no segment has closed — **deliberately not live**
`LIVE`	A segment exists
`ENDED`	The broadcaster stopped. Also where an abandoned schedule lands, silently — a stream that announces its own abandonment is worse than one that quietly started late
`PROCESSING`	Recording handed to File Thunder
`REPLAY_READY`	Replayable; `replayMediaId` populated, and **`playbackUrl` is replaced** — see below
`FAILED`	Setup failed, or the recording could not be processed

> ⚠️ **`playbackUrl` changes meaning at `REPLAY_READY`.** While live it points at SRS; once the replay is ready it is **replaced** with a CDN address on a different host entirely. Always re-read `playbackUrl` from the stream rather than caching the live one — a player still pointed at the live URL after the broadcast ends is pointed at nothing.
>
> ```
> LIVE          https://stream.dev.nexgate.co/live/<ingestKey>.m3u8
> REPLAY_READY  https://cdn-…/nexgate-
> public/streams/<owner>/<mediaId>/hls/master.m3u8
> ```
>
> The replay is a proper HLS ladder with a master playlist, so a player may pick a rendition; the live URL is a single variant.

`REPLAY_READY` rather than `VOD_READY` on purpose: "video on demand" is wrong for `AUDIO_RADIO`, which has no video, and a state named after a format the mode does not produce quietly teaches the next reader something false.

Modes

`StreamMode`	Shape	SRS app
`LIVE_VIDEO`	One broadcaster, video and audio (default)	`live`
`AUDIO_RADIO`	One broadcaster, audio only	`radio`

Radio publishes into its **own SRS app**, which is what scopes the 32 kbps transcode to the streams that actually need it. Mode is chosen at creation and **never changed** — switching mid-session renegotiates every participant's media and resets every HLS player.

(Spaces used to be a third mode. They now have their own entity and their own endpoints — see Part 4.)

`POST` /api/v1/live/streams — Create

Setup only. Nothing is live and nothing costs anything — the host may fill it in, leave, and come back.

Request (`CreateStreamRequest`):

Field	Type	Required	Description

`title`	string	**yes**	Unlike a call, a stream is something people find and choose
`description`	string	no	
`coverMediaId`	UUID	no	
`scheduledFor`	ISO-8601	no	Absent → `DRAFT` (about to start). Future → `SCHEDULED`, which announces itself and reminds people
`mode`	enum	no	`LIVE_VIDEO` if unset

Returns `StreamResponse`.

`POST` /api/v1/live/streams/{streamId}/ingest-key — Get broadcast credentials

Called from the device-check screen, once the host is actually ready. **This is the only place an ingest key is ever returned.** Host only.

Response:

```
{
  "streamId": "…",
  "ingestKey": "…",
  "expiresAt": "2026-09-14T08:36:00Z",
  "whipUrl": "https://stream.dev.nexgate.co/rtc/v1/whip/?app=live&stream=<ingestKey>",
  "rtmpUrl": "rtmp://stream.dev.nexgate.co:1935/live/<ingestKey>"
}
```

The stream **name** a publisher uses **is the key itself** — that is what lets SRS hand it to us for checking without any other channel.

Which URL to publish to:

You are publishing from	Use	Why
A phone or browser	`whipUrl`	WebRTC; survives a mobile network and needs no extra app

You are publishing from	Use	Why
OBS or a desktop encoder	`rtmpUrl`	what those tools speak

> **Take both URLs from this response. Never hardcode them.** They differ per environment, and until recently they were laptop addresses — a client that pinned them would have told every broadcaster to publish to their own device.

> **The key expires (~30 minutes) and is single-purpose.** Fetch it on the device-check screen, immediately before publishing — not when the stream is created. If the host sets up and then wanders off, ask again.

`POST` /api/v1/live/streams/{streamId}/end — End the broadcast

Host only. Returns `StreamResponse` with `state: ENDED`.

`PUT` /api/v1/live/streams/{streamId}/reminder — Remind me

Param	In	Default
`wanted`	query	`true`

A `PUT` with `wanted=false` rather than a `DELETE` route, because "remind me" is one toggle in the UI.

`GET` /api/v1/live/streams/{streamId} — Read a stream

`StreamResponse`

Field	Type	Notes
`id`	UUID	
`mode`	enum	
`state`	enum	
`hostUserId`	UUID	
`title`, `description`	string	
`coverMediaId`	UUID	
`scheduledFor`	ISO-8601	
`playbackUrl`	string	**null unless `LIVE` or `REPLAY_READY`**
`audioOnly`	boolean	
`startedAt`, `endedAt`	ISO-8601	
`peakViewers`	int	
`replayMediaId`	UUID	

Note what is absent: `ingestKey`. It is a broadcast credential, returned only from the endpoint the host calls to get one, and never carried on an object that viewers also read.

`playbackUrl` is gated on **state**, not on `replayMediaId` being set — that id is stored the moment the recording is handed to File Thunder, so during `PROCESSING` it exists but points at nothing playable yet.

Stream refusal codes

Code	HTTP	When
`NOT_FOUND`	404	no such stream
`TITLE_REQUIRED`	409	no title
`NOT_HOST`	409	host-only verb
`NOT_SCHEDULED`	409	the stream is not in a schedulable state

Code	HTTP	When
`STREAM_FINISHED`	409	acting on a stream that is over
`NO_ACCOUNT`	409	no account behind the caller
`ACCOUNT_TOO_NEW`	409	anti-abuse: account age gate on going live
`AGE_RESTRICTED`	409	

Part 2 — Live comments

`api/v1/live/streams/{streamId}/comments`

A write endpoint and a read endpoint, and the difference between them is the whole design.

```
viewer ──POST──> origin        (per viewer, rate limited, moderated)

10 000 viewers ──GET──> [ CDN, 2s cache ] ──one read──> origin
```

The write is per viewer and rate limited. The read is the **same response for everybody**, cached at the CDN for two seconds — one origin read serving thousands. **That is why it takes no parameters:** a `since` cursor would give every viewer a unique URL and turn the cache off. The client drops what it has already rendered using the per-stream monotonic `seq`.

`POST` …/comments — Post a comment

Request: `{ "body": "…" }`

Refuses unless the stream is `LIVE`. Then moderation runs:

Verdict / code	Message
`NOT_LIVE`	This stream is not live
`MUTED`	You cannot comment on this stream
`TOO_FAST`	Slow down
`SLOW_MODE`	Slow mode is on
`FILTERED`	That comment was not posted
`TOO_LONG`	That comment is too long

Verdict / code	Message
`EMPTY`	Say something

On success returns the posted `LiveComment` **to the poster**, even though the read window may sample it away for everyone else. A viewer always sees their own comment — it is what makes the feature feel alive when a comment reached nobody.

`GET` …/comments — Read the overlay

No parameters. Identical for every viewer. `Cache-Control: public, max-age=2`**.**

Deliberately **not** wrapped in the platform envelope — the payload is the comments:

```
[
  { "seq": 412, "userId": "…", "name": "Mama Yoyo", "body": "bei gani?",
    "at": 1755680000000, "system": false, "event": null, "refId": null }
]
```

Field	Type	Notes
`seq`	long	Per-stream, monotonic. The client drops what it already rendered
`userId`, `name`	string	The name is carried inline because a cached read cannot resolve names per viewer
`body`	string	
`at`	long	epoch millis
`system`	boolean	System events share the stream — someone joined, the host pinned something
`event`	string	**A key, not a sentence.** The server writing English would ship untranslatable text into a Swahili UI
`refId`	string	What the event refers to

A comment is deliberately **not** shaped like a chat message: no conversation, no recipient, no delivery guarantee, no id the sender can retry against.

Host tools

Method	Path	Params
`PUT`	…/comments/mute/{userId}	`muted` (default `true`)
`PUT`	…/comments/slow-mode	`seconds` (0 turns it off)

Both host-only; a non-host gets `409 NOT_HOST`.

Part 3 — Attachments (the commerce shelf)

`api/v1/live/streams/{streamId}/attachments`

Host verbs are attach, pin, unpin and remove; the only viewer verbs are reading the shelf and tapping through. **Pinning is a live act** — most attachments arrive while the host is talking, not at setup.

`AttachmentState`	Meaning
`SHELF`	Attached and browsable, not on screen. Unbounded
`PINNED`	On screen right now. **Exactly one per stream** — that is what tells a viewer arriving mid-stream which thing is being talked about now
`REMOVED`	Taken down. Kept rather than deleted so a replay still shows what was on screen at the time

`AttachmentType`	
`PRODUCT`	something to buy
`SHOP`	a duka to visit — "this is where I get these"
`EVENT`	
`TICKET`	also bought, so attribution is not product-only
`POST`	an earlier post, for context
`LINK`	

All types are equal — attach any number, of any type, from any shop, at any point in the stream's life.

Method	Path	Who	Description
`POST`	…/attachments	host	Body `{ "type": "PRODUCT", "targetId": "…" }`

Method	Path	Who	Description
`GET`	.../attachments	anyone	Everything attached and not removed, in the order it was added
`PUT`	.../attachments/{attachmentId}/pin	host	Puts it on screen
`DELETE`	.../attachments/pin	host	Clears the pin
`DELETE`	.../attachments/{attachmentId}	host	Removes it
`POST`	.../attachments/{attachmentId}/tap	viewer	Records a tap-through

`AttachmentResponse`

Field	Notes
`id`, `type`, `targetId`, `state`	
`streamOffsetMs`	The client applies a pin at this **playback position** rather than on arrival — the whole point of recording it
`pinnedAt`	
`taps`	Counted **apart from orders** — interest is not money, and the post-stream figures keep them separate so the number a seller reads means something

---

Part 4 — Spaces

`api/v1/live/spaces` — many speakers, audio, an HLS audience.

> This part is the **complete** Spaces reference. It was a separate document until 2026-09-14; everything from it is here.

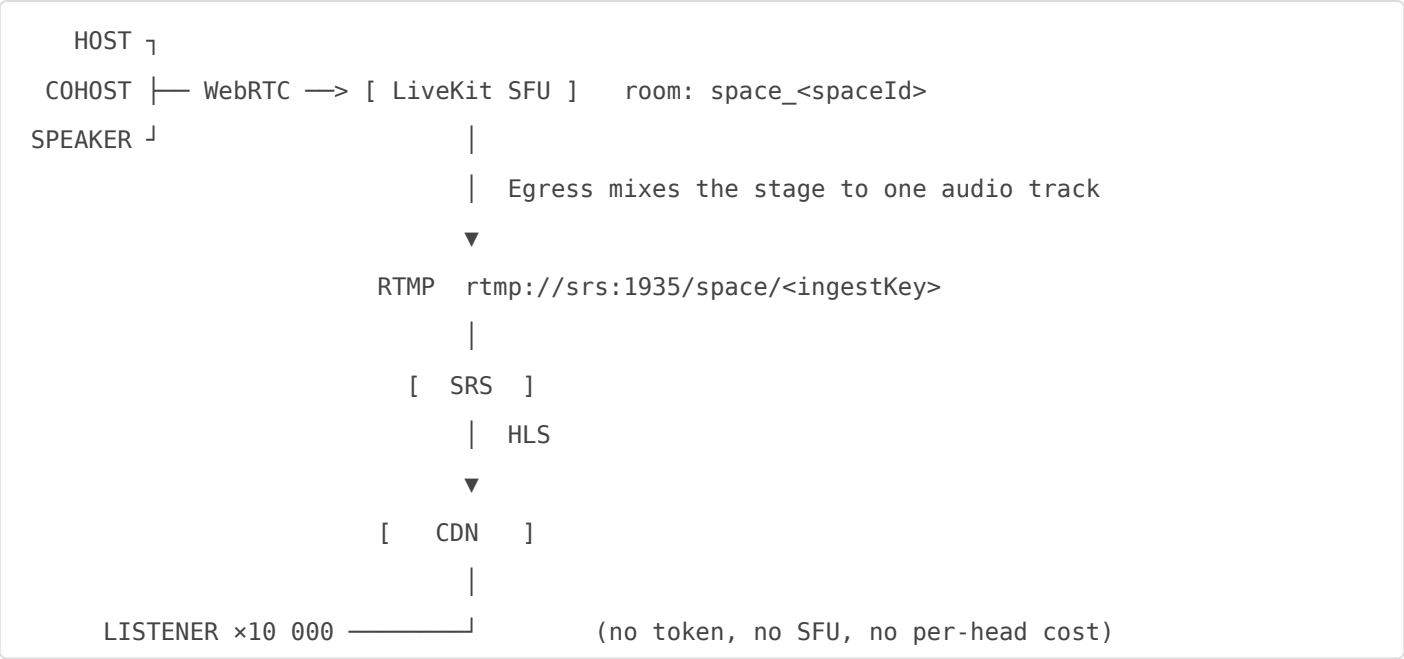
Why a Space is not a stream

A Space started life as VP Live's third `StreamMode`, which made it share a state machine, ingest keys and scheduling with Live and Radio. It now has its own table and its own endpoints, because:

- it is a **two-way conversation**, so it belongs with the other two-way features in `chat_system`;
- sharing the row meant `chat_system` importing `live_mng` — the boundary crossing the split removes.

What it **keeps** from that heritage is the *shape*, and deliberately so: the stage is WebRTC and the audience is HLS, so a Space still needs an ingest key (Egress publishes to SRS with it) and a playback URL (what listeners actually play). Those are not chat concepts; they are **the price of an audience that does not cost per head**.

The cost model — the one diagram to present

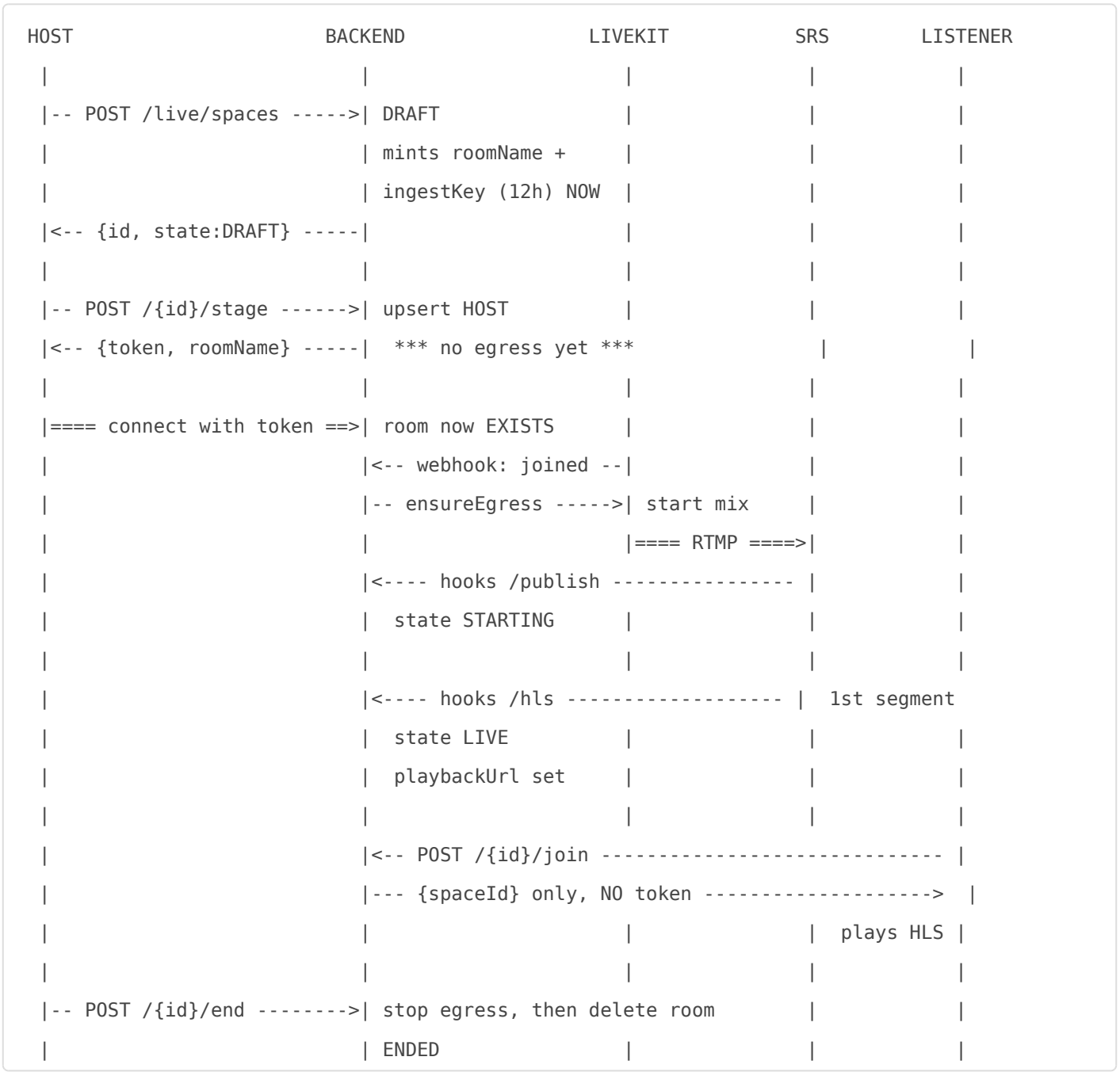


An SFU costs **linearly per subscriber**. An audience of ten thousand on LiveKit would need ten thousand SFU downstreams; on HLS it is one origin read fanned out by the CDN. So listeners must never become LiveKit participants — and the enforcement is simply that **they are never given a token**.

`SpaceRole`	**On the SFU?**	**May change the stage?**
`HOST`	yes	yes
`COHOST`	yes	yes
`SPEAKER`	yes	no
`LISTENER`	**no** — HLS via CDN	no

Stage cap: **10** (`app.live.space-stage-limit`). A product limit, not an infrastructure one — Egress was measured carrying 10+ concurrent Spaces. A conversation with twenty people talking is not a conversation.

Lifecycle — and the trap in the middle of it

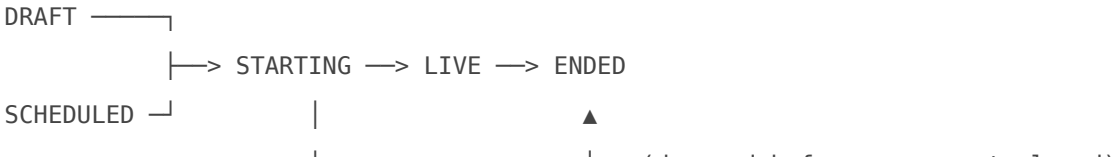


The trap, and it cost a debugging session: the mix is **not** started at `openStage`. Egress cannot mix a room that does not exist, and a LiveKit room does not come into being until the first participant connects — which has not happened yet at `openStage`, because we are still minting the

token they will use to connect. Starting egress there returns **404 every time**. So `ensureEgress` runs off the LiveKit *participant-joined webhook* instead, and is idempotent, so every later joiner is a no-op.

If egress refuses, the log says it plainly: "*speakers will hear each other and nobody else will.*" That is the exact failure signature — a stage that works and an audience that gets nothing.

State machine



State	Meaning	Set by
`DRAFT`	Being set up. The room does not exist yet and nothing costs anything	`create`
`SCHEDULED`	Booked for later	`create` with `scheduledFor`
`STARTING`	The mix is reaching SRS, but no segment has closed — a listener has nothing to play yet. **Deliberately not `LIVE`**	SRS `/publish`
`LIVE`	A segment exists. The first moment a listener can actually hear it	SRS `/hls`
`ENDED`	Over	host `end`, or SRS `/unpublish`

`onUnpublish` counts `STARTING` as well as `LIVE`: a stage that connected and dropped before any segment closed still ended.

> `SpaceState` also declares `PROCESSING`, `REPLAY_READY` and `FAILED`. **No code path sets them today** — see *Known gaps*.

Space endpoints

`api/v1/live/spaces`

Method	Path	Token?	Who

`POST`	`/live/spaces`	no	anyone
`GET`	`/live/spaces/{spaceId}`	no	anyone
`POST`	`/live/spaces/{spaceId}/stage`	**yes**	host
`POST`	`/live/spaces/{spaceId}/join`	**no, on purpose**	anyone
`PUT`	`/live/spaces/{spaceId}/hand`	no	participant
`PUT`	`/live/spaces/{spaceId}/speakers/{userId}`	**yes**	host / co-host
`DELETE`	`/live/spaces/{spaceId}/speakers/{userId}`	no	host / co-host
`DELETE`	`/live/spaces/{spaceId}/me`	no	participant
`GET`	`/live/spaces/{spaceId}/participants`	no	anyone
`POST`	`/live/spaces/{spaceId}/end`	no	host

`POST` /api/v1/live/spaces — Create

Request:

Field	Type	Required	Description
`title`	string	**yes**	
`description`	string	no	
`scheduledFor`	ISO-8601	no	Present → `SCHEDULED`; absent → `DRAFT`

```
curl -s -X POST $BASE/api/v1/live/spaces \
  -H "Authorization: Bearer $TOKEN" -H 'Content-Type: application/json' \
  -d '{"title":"Friday pricing talk"}'
```

The host is **always a person** — a shop cannot hold a conversation.

`roomName` and `ingestKey` are minted **up front**, at creation: Egress needs the key the moment the stage opens, and unlike a phone there is no device-check screen to ask on. The key lives **12 hours** and is single-purpose, so a leaked key is not a standing right to publish into this Space.

`GET` /api/v1/live/spaces/{spaceId} — Read

Response (`data`):

Field	Type	Notes
`id`	UUID	
`state`	enum	
`hostUserId`	UUID	
`title`, `description`	string	
`scheduledFor`	ISO-8601	
`playbackUrl`	string	**null unless** `LIVE`
`startedAt`, `endedAt`	ISO-8601	

Note what is absent: `ingestKey`, `roomName` **and** `egressId`**.** They are broadcast internals and never appear on an object listeners read.

`POST` /api/v1/live/spaces/{spaceId}/stage — Open the stage

Host only. Registers the host as `HOST` and returns a LiveKit token.

Response:

```
{
  "spaceId": "…",
  "roomName": "space_<spaceId>",
  "token": "…",
  "url": "wss://livekit.dev.nexgate.co"
}
```

> `url` **is where to spend the token**, and it arrives beside every token — here and on promotion. **Read it from the response; never hardcode it.** It was missing from both until 2026-09-14, so a speaker held a credential with no server to use it against. If your client hardcoded an address to work around that, remove it.

This does not make the Space `LIVE`. SRS does that when the mixed audio reaches it — a room with a host and no audio is not a broadcast, and marking it live hands listeners a playlist that 404s.

`POST` /api/v1/live/spaces/{spaceId}/join — Join the audience

Returns no connection details, deliberately:

```
{ "spaceId": "…" }
```

The listener plays the `playbackUrl` from `GET /live/spaces/{spaceId}` like any other HLS viewer. A token here is precisely what would put the audience on the SFU.

`PUT` /api/v1/live/spaces/{spaceId}/hand — Raise / lower a hand

Param	In	Required
`raised`	query	**yes** (`true` / `false`)

Refuses with `ALREADY_SPEAKING` if you are already on stage, and `NOT_IN_SPACE` if you never joined.

`PUT` /api/v1/live/spaces/{spaceId}/speakers/{userId} — Promote

Host or co-host. Sets the target to `SPEAKER`, clears their raised hand, and returns **their** token:

```
{ "userId": "…", "token": "…", "url": "wss://livekit.dev.nexgate.co" }
```

The riskiest interaction in the feature: the token is a **permission, not a connection**. Crossing from HLS playback to a LiveKit publish without a gap in audio is the client's problem.

`DELETE` /api/v1/live/spaces/{spaceId}/speakers/{userId} — Demote

Back to `LISTENER`. Refuses `CANNOT_DEMOTE_HOST`.

The token is not revoked, because a LiveKit token cannot be revoked — the same limit calls hit with the camera rule. What ends their turn is the client disconnecting; recording the demotion is what makes the *next* promotion valid rather than a duplicate.

`DELETE` /api/v1/live/spaces/{spaceId}/me — Leave

Sets `leftAt` and lowers the hand. **The row is kept, not deleted** — a Space is a conversation, and who spoke in it is what moderation needs to answer later; deleting the row deletes the answer.

Rejoining after a dropped connection **must not demote a speaker back into the audience**, so the upsert only ever raises a `LISTENER`, never lowers a speaker.

`GET` /api/v1/live/spaces/{spaceId}/participants — Who is here

Everyone who has not left. The envelope `message` is `"<n> in this Space"`.

```
[ { "userId": "…", "role": "SPEAKER", "handRaised": false, "joinedAt": "…" } ]
```

`POST` /api/v1/live/spaces/{spaceId}/end — End it

Host only. Idempotent — ending an `ENDED` Space returns it unchanged.

Order matters: **stop the mix first, then delete the room.** Deleting the room drops the stage, and an egress still running against a room that no longer exists is a job nobody will ever stop.

The ingest key is deliberately **not cleared** on end: a recording callback arrives after the Space ends and is named after the key SRS received. Nothing is weakened, because a usable key requires `acceptsPublish()`, which an `ENDED` Space fails.

Refusal codes

Code	When
`NOT_FOUND`	no such Space
`NOT_HOST`	host-only verb (`stage`, `end`)
`NOT_OPEN`	the Space has ended; it will not accept a publisher
`NOT_IN_SPACE`	you (or the target) never joined
`NOT_STAGE_CONTROL`	only the host or a co-host can change the stage
`STAGE_FULL`	at the stage limit (10)
`ALREADY_SPEAKING`	raising a hand while already on stage
`CANNOT_DEMOTE_HOST`	the host holds the Space

Space server-to-server (internal)

Neither of these is for clients. **They are what actually drive Space state.**

LiveKit webhooks ? `SpaceRoomHandler`

Routed by room-name prefix: **calls own `call_*`, Spaces own `space_*`.** Without that split every Space event would reach `CallWebhookService`, which resolves each one against a `CallEntity` and would find nothing — quietly, for every participant of every Space.

Event	Effect
participant joined	`ensureEgress` — the earliest moment Egress can be asked to mix. Idempotent
participant left	`leave(user, space)`. Only stage members are LiveKit participants, so **this never fires for a listener**
room finished	Logged only. The Space's state still belongs to SRS
track published	A non-`AUDIO` track is logged as a **client bug** — Spaces are audio only

SRS hooks ? `SpaceSrsHandler` (app `space`)

The publisher here is **never a person**: it is LiveKit Egress pushing the mixed stage in as RTMP. SRS cannot tell the difference and does not need to — the ingest key is the credential either way, which is what lets a Space go `LIVE` by the same rule as every broadcast: **because media arrived, not because an app said so.**

Hook	Effect
`/publish`	Validates the key; `STARTING`. Returns a refusal if the key authorises nothing
`/hls`	First segment closed → `LIVE`, `playbackUrl` set
`/unpublish`	The mix stopped → `ENDED`
`/dvr`	**Logged, not published.** Recording a Space is a consent question a broadcast does not raise, and is not built. The file exists on disk for moderation only

---

Space configuration

Property	Local default	Staging	Meaning
`app.live.space-stage-limit`	`10`	`10`	Concurrent speakers
`app.live.rtmp-ingest-url`	`rtmp://srs:1935`	unchanged	Where Egress publishes the mix. **Internal** — never handed to a client
`app.live.playback-base-url`	`http://127.0.0.1:8080`	`https://stream.dev.nexgate.co`	HLS base; playlist is `/space/<ingestKey>.m3u8`

Property	Local default	Staging	Meaning
`app.livekit.url`	`ws://127.0.0.1:7880`	`wss://livekit.dev.nexgate.co`	SFU websocket — the `url` returned with every token

Ingest key: 24 random bytes, URL-safe Base64, unpadded. Valid 12 hours.

Schema note

Enum columns are **VARCHAR, not Postgres enums**. Hibernate freezes a `CHECK` constraint around an enum column when the table is created and never revisits it, so a value added later is refused at `INSERT` — **a trap that has now cost this project four separate times**, most recently when `VOICE_NOTE` was added to chat. Adding an enum value means updating the constraint in the same change.

Known gaps

Honest list, from reading the package — worth knowing before the demo so nobody asks about them from the floor.

Gap	Detail
`COHOST` **cannot be assigned**	`requireStageControl` honours it, but no endpoint or code path ever *sets* the role. `promote` grants `SPEAKER`. Today, only the host can control the stage
`SCHEDULED` **never opens itself**	Nothing schedules or transitions it — no job, no announcer. The host simply opens the stage whenever they like. Unlike streams, which have `StreamScheduleJob`
No announcements or notifications	Nothing outside `space_mng` references Spaces at all. No follower announce, no reminders, no push
`PROCESSING` / `REPLAY_READY` / `FAILED` **are unreachable**	Declared on `SpaceState`, never set. Replay is not built — `/dvr` deliberately does not publish
No comments on a Space	Live comments are a stream feature (`/live/streams/{id}/comments`); Spaces have no equivalent overlay
Demotion cannot force a disconnect	LiveKit tokens are not revocable; a demoted speaker stops publishing only when their client cooperates

---

Part 5 — SRS hooks (internal)

`api/v1/live/hooks` — SRS → backend. **Not for clients.** These are what actually drive stream state:

Hook	Effect
`POST` /live/hooks/publish	Validates the ingest key; `STARTING`. A stream that has ended must not accept a publisher reconnecting with an old key
`POST` /live/hooks/hls	First segment closed → `LIVE`, announce, notify
`POST` /live/hooks/unpublish	Publisher gone → `ENDED`
`POST` /live/hooks/dvr	Recording ready → `PROCESSING` → `REPLAY_READY`

Configuration

Property	Default	Meaning
`app.live.publisher-rtmp-url`	`rtmp://127.0.0.1:1935`	**Public** — handed to a broadcaster as `rtmpUrl`
`app.live.publisher-whip-url`	`http://127.0.0.1:1985`	**Public** — handed to a broadcaster as `whipUrl`
`app.live.playback-base-url`	`http://127.0.0.1:8080`	**Public** — where viewers fetch HLS
`app.live.rtmp-ingest-url`	`rtmp://srs:1935`	**Internal** — Egress → SRS only
`app.live.recording-base-url`	`http://srs:8080`	**Internal** — File Thunder fetching a finished recording

> **Public and internal are not interchangeable.** The first three are addresses a phone must be able to reach; the last two never leave the server network. Handing a client an internal one tells it to publish to a host that does not exist outside the cluster. Space configuration is in Part 4.

SRS treats Egress as an ordinary broadcaster, so the ingest key is the credential exactly as it is for a phone publishing a stream.

Revision #1
Created 27 November 2026 08:20:28 by Admin Qbit
Updated 27 November 2026 08:21:25 by Admin Qbit