Live Streaming (VP Live) API Documentation

Author: Josh S. Sakweli, Backend Lead Team **Last Updated**: 2026-08-20 **Version**: v1.0

Base URL: `http://localhost:8765/api/v1` (local) — `server.port=8765`

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 1 — Streams

`api/v1/live/streams`

Why there is no "go live" button

```
|-- 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
`FAILED`	Setup failed, or the recording could not be processed

`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`

`StreamMode`	Shape	SRS app
`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": "2026-08-20T…",
  "whipUrl": "http://127.0.0.1:1985/rtc/v1/whip/?app=live&stream=<ingestKey>",
  "rtmpUrl": "rtmp://127.0.0.1/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.

`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`

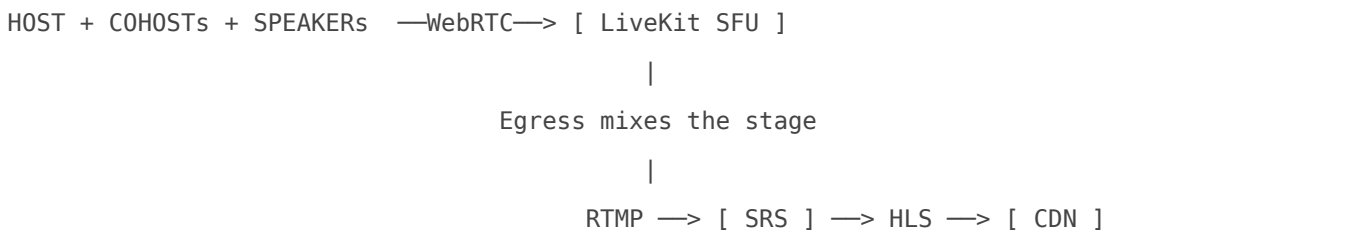
Field	Notes
`id`, `type`, `targetId`, `state`	
`streamOffsetMs`	The client applies a pin at this **playback position** rather than on arrival — the whole point of recording it
`pinnedAt`	
`taps`	Counted **apart from orders** — interest is not money, and the post-stream figures keep them separate so the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

---

Part 4 — Spaces

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

The cost model, which is the whole design



```
                                                    |
LISTENERS ──────────────────────────────────────────┘
```

Note which endpoints return a token and which do not. Opening the stage and being promoted return one; joining as a listener never does. *That asymmetry IS the cost model* — a token is what would put the audience on the SFU. An audience of ten thousand on LiveKit would need ten thousand SFU downstreams; on HLS it is one CDN origin.

`SpaceRole`	**On the SFU?**
`HOST`	yes
`COHOST`	yes — may also change the stage
`SPEAKER`	yes
`LISTENER`	**no** — HLS through the CDN like any other viewer

Stage cap: **10** by default (`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.

`SpaceState` mirrors the stream states: `DRAFT`, `SCHEDULED`, `STARTING`, `LIVE`, `ENDED`, `PROCESSING`, `REPLAY_READY`, `FAILED`. Same rule — `LIVE` **happens because SRS reported the mixed audio arrived**, not because the host's app said so.

Endpoints

Method	Path	Returns a token?	Description
`POST`	/live/spaces	no	Body `{ title, description?, scheduledFor? }`
`GET`	/live/spaces/{spaceId}	no	`playbackUrl` is null until `LIVE`
`POST`	/live/spaces/{spaceId}/stage	**yes**	Host opens the stage. **Does not make the Space** `LIVE` — SRS does that
`POST`	/live/spaces/{spaceId}/join	**no, on purpose**	Join as a listener; play the HLS URL
`PUT`	/live/spaces/{spaceId}/hand	no	`raised=true
`PUT`	/live/spaces/{spaceId}/speakers/{userId}	**yes**	Promote to speaker
`DELETE`	/live/spaces/{spaceId}/speakers/{userId}	no	Move back to the audience

Method	Path	Returns a token?	Description
`DELETE`	/live/spaces/{spaceId}/me	no	Leave
`GET`	/live/spaces/{spaceId}/participants	no	`userId`, `role`, `handRaised`, `joinedAt`
`POST`	/live/spaces/{spaceId}/end	no	

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

Space refusal codes

Code	When
`NOT_FOUND`	no such Space
`NOT_OPEN`	the Space is not accepting people
`NOT_IN_SPACE`	you are not in it
`NOT_HOST`	host-only verb
`NOT_STAGE_CONTROL`	only the host or a co-host can change the stage
`STAGE_FULL`	at the stage limit
`ALREADY_SPEAKING`	already on stage
`CANNOT_DEMOTE_HOST`	

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.space-stage-limit`	`10`	Concurrent speakers in a Space
`app.live.rtmp-ingest-url`	`rtmp://srs:1935`	Where Egress publishes the mixed stage

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 20 August 2026 16:51:00 by Admin Qbit
Updated 20 August 2026 16:51:15 by Admin Qbit