VP Live & VP Audio Spaces

Live Streaming Architecture

NexGate / QBIT SPARK | Version 1.0 *SRS · HLS · LiveKit · VP Live Video · VP Audio Radio · VP Audio Spaces*

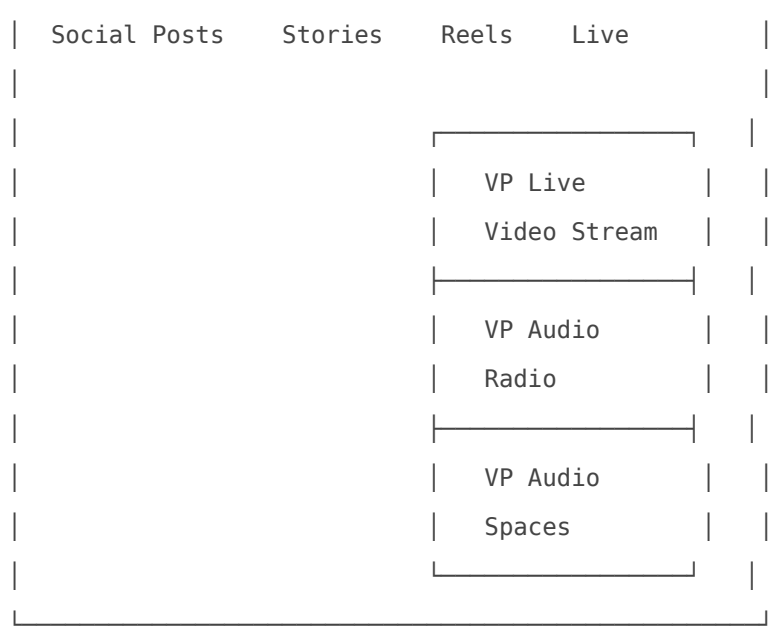
Table of Contents

---

1. Overview

VP Live and VP Audio Spaces live under **VP Feed** — the social pillar of NexGate. They are not separate products. They are the live expression layer of the social platform — where creators, merchants, and communities connect with their audiences in real time.

```
VP Feed
┌──────────────────────────────────────────────────┐
│                                                  │
```



All three modes share the same infrastructure foundation: SRS for ingest and transcoding, Cloudflare CDN for delivery, Ejabberd MUC for live chat, File Thunder for VOD processing, and Spring Boot for stream management and business logic.

---

2. VP Live vs VP Audio — Key Differences

	VP Live (Video)	VP Audio Radio (Radio/Podcast)	VP Audio Spaces (Twitter Spaces)
Broadcasters	1	1	Multiple (up to 30)
Viewers	Unlimited	Unlimited	Unlimited listeners
Direction	One way	One way	Multi-speaker
Broadcaster transport	RTMP (video+audio)	RTMP audio (audio only)	WebRTC (LiveKit)
Listener transport	HLS video (adaptive)	HLS audio (adaptive)	HLS audio (listeners) WebRTC (speakers)
Latency	6-15 seconds	6-15 seconds	Speakers: <200ms Listeners: 6-15s
Bandwidth broadcaster	High (2-4 Mbps)	Very low (128 kbps)	Low (speakers) Very low (listeners)
Bandwidth	Medium	Very low	Very low

```
listener          (300kbps-2Mbps) (32-128 kbps)     (32-128 kbps)
Works on 2G?      □ No            □ Yes             □ Listeners yes
Live chat         Ejabberd MUC    Ejabberd MUC      Ejabberd MUC
Raise hand        □               □                 □
VOD after         □ File Thunder  □ File Thunder    □ File Thunder
New infra         SRS             SRS               SRS + LiveKit
```

---

3. How Live Streaming Works

The Core Pattern — RTMP ? HLS ? CDN

```
Broadcasting (sending):
  Broadcaster's phone records camera + mic
  App encodes: H.264 video + AAC audio
  App streams via RTMP protocol to SRS server
  One stream upload from broadcaster

Processing (server):
  SRS receives RTMP stream
  FFmpeg transcodes to multiple quality variants
  Packages into HLS format (2-second chunks)
  Writes chunks to MinIO storage every 2 seconds

Delivery (viewing):
  Cloudflare CDN pulls chunks from MinIO
  Caches chunks at edge nodes globally
  Viewers request HLS playlist → adaptive player picks quality
  10,000 viewers = 10,000 CDN requests, NOT 10,000 SRS requests
  SRS barely notices the viewer count

Why HLS and not WebRTC for viewers:
  WebRTC to viewers: broadcaster uploads N streams (one per viewer)
  HLS via CDN:       broadcaster uploads 1 stream → CDN serves all
  At 10,000 viewers: WebRTC = impossible, HLS = trivial
```

HLS — What It Actually Is

```
HLS (HTTP Live Streaming) — Apple's open standard

SRS generates:
  master.m3u8         → playlist of all quality variants
  360p/playlist.m3u8  → playlist for 360p variant
  360p/seg_000.ts     → 2-second video chunk
  360p/seg_001.ts     → next 2-second chunk
  720p/playlist.m3u8
  720p/seg_000.ts
  ...

master.m3u8 looks like:
  #EXTM3U
  #EXT-X-STREAM-INF:BANDWIDTH=400000,RESOLUTION=640x360
  360p/playlist.m3u8
  #EXT-X-STREAM-INF:BANDWIDTH=1500000,RESOLUTION=1280x720
  720p/playlist.m3u8

Player (ExoPlayer / AVPlayer):
  Downloads master.m3u8 first
  Measures current network speed
  Picks 360p if on 3G → plays seg_000.ts → seg_001.ts → ...
  Switches to 720p if network improves → seamless
  All automatic — zero app code needed for quality switching
```

---

4. VP Live — Video Streaming

Full Architecture

```
[Broadcaster Phone]
     |
     | RTMP stream
     | rtmp://stream.nexgate.com/live/{streamKey}
     | H.264 video + AAC audio
     | ~2-4 Mbps upload
     ▼
[SRS Media Server]
```

```
      │
      ├── Validates stream key:
      │     POST /internal/stream/validate
      │     { streamKey: "abc123" }
      │     Spring Boot: ✅ allow or ❌ reject
      │
      ├── Receives raw RTMP stream
      │
      ├── FFmpeg transcoding (real-time):
      │     1080p H.264 → 3 Mbps  (WiFi viewers)
      │     720p  H.264 → 1.5 Mbps (4G viewers)
      │     480p  H.264 → 600 kbps (3G viewers)
      │     360p  H.264 → 300 kbps (2G viewers)
      │
      ├── Package as HLS:
      │     Segment every 2 seconds
      │     live/{streamKey}/master.m3u8
      │     live/{streamKey}/360p/seg_NNN.ts
      │     live/{streamKey}/720p/seg_NNN.ts
      │
      └── Write to MinIO: nexgate-live bucket
            New segments every 2 seconds
      │
      ▼
[Cloudflare CDN]
      │ Pulls from MinIO automatically
      │ Caches at edge (Nairobi edge closest to EA)
      │ Short TTL: 10 seconds (live content)
      │
      ▼
[Viewers — ExoPlayer (Android) / AVPlayer (iOS)]
     Requests master.m3u8
     Player picks quality based on network
     Downloads .ts segments every 2 seconds
     Seamless adaptive quality switching
```

Stream Key Validation Flow

```
Broadcaster taps "Go Live" in app
     |
     ▼ POST /live/start
Spring Boot:
  Generate unique stream key
  Store in DB:
    stream_key: "abc123"
    user_id: usr-kibuti
    status: PENDING
    created_at: now
  Return stream key to app
     |
App connects RTMP:
  rtmp://stream.nexgate.com/live/abc123
     |
SRS receives connection
     |
     ▼ POST /internal/stream/validate (SRS webhook)
Spring Boot checks:
  Key exists? □
  User account active? □
  User has live permission? □
  No other active stream for this user? □
  → 200 OK → SRS allows stream
  → Update DB: status: LIVE, started_at: now
  → Notify followers via FCM:
      "Kibuti anastreamu sasa! Tazama live"
  → Create Ejabberd MUC room:
      live-abc123@conference.nexgate.com
```

Broadcaster App — What Mobile Dev Implements

```
Android library: rtmp-rtsp-stream-client-java
iOS library: HaishinKit (Swift)

Steps for broadcaster app:
  1. GET /live/start → receive stream key
```

```
  2. Initialize camera + microphone
  3. Connect RTMP to stream.nexgate.com/live/{key}
  4. Start streaming — library handles everything:
       H.264 encoding (hardware)
       AAC audio encoding
       RTMP packet framing
       Network reconnection on drop
  5. Show: viewer count (from Redis via REST poll)
           live comments (from Ejabberd MUC via WS)
           duration timer
  6. Tap End → POST /live/end → cleanup

Adaptive upload bitrate:
  Library monitors upload speed
  Reduces video quality if upload struggles
  Broadcaster's bad network → lower quality for viewers
  Never drops stream if avoidable
```

Viewer App — What Mobile Dev Implements

```
Android: ExoPlayer (Google's official video player)
iOS: AVPlayer (built into iOS, zero setup)

Steps for viewer app:
  1. GET /live/{streamId}/url
     Response: { masterUrl, viewerCount, startedAt }
  2. Feed masterUrl to ExoPlayer/AVPlayer
  3. Player handles everything automatically:
       Downloads master.m3u8
       Picks quality based on network
       Downloads segments every 2s
       Switches quality up/down seamlessly
  4. Join Ejabberd MUC room → show live comments
  5. Player shows: loading → buffering → playing

That is genuinely all the viewer needs to implement.
HLS + ExoPlayer/AVPlayer is the easiest viewer experience
to build in all of mobile development.
```

---

5. VP Audio Radio — One Broadcaster Many Listeners

Why Audio Radio Matters for EA

```
VP Live video:
  Broadcaster needs: 2-4 Mbps upload
  Viewer needs:      300kbps minimum
  Data cost viewer:  ~900MB per hour at 360p
  Works on:          4G and strong 3G only

VP Audio Radio:
  Broadcaster needs: 64-128 kbps upload
  Listener needs:    32 kbps minimum
  Data cost listener: ~15MB per hour at 32kbps
  Works on:          2G, Edge, any connection

For a farmer in rural Tanzania with 2G:
  VP Live video → impossible, too expensive
  VP Audio Radio → accessible, affordable

Use cases:
  Live podcast / commentary
  Religious broadcasts (huge in EA)
  Political discussions
  Community announcements
  Sports commentary
  Language learning sessions
  Business webinars (audio only)
```

Architecture — Same SRS, Audio Only

```
[Broadcaster Phone]
     |
     | RTMP audio only (no video track)
     | AAC codec, 128 kbps
```

```
     | rtmp://stream.nexgate.com/audio/{streamKey}
     ▼
[SRS Media Server]
     |
     ├── Same validation flow as VP Live
     |
     ├── FFmpeg transcoding (audio only):
     |     AAC 128 kbps → good network listeners
     |     AAC  64 kbps → 3G listeners
     |     AAC  32 kbps → 2G listeners
     |
     ├── Package as HLS audio:
     |     audio/{streamKey}/master.m3u8
     |     audio/{streamKey}/128k/seg_NNN.aac
     |     audio/{streamKey}/32k/seg_NNN.aac
     |
     └── Write to MinIO: nexgate-live bucket
     |
     ▼
[Cloudflare CDN]
     |
     ▼
[Listeners — ExoPlayer / AVPlayer]
     HLS audio playlist
     Adaptive bitrate: 128k → 32k automatically
     Same player, same code — just no video surface
```

Codec Choice — AAC Not Opus

```
Why AAC for HLS audio radio (not Opus):

Opus is better quality at low bitrates — true
But HLS has a compatibility requirement:
  Apple mandates AAC for HLS audio
  AVPlayer on iOS does not support Opus in HLS
  Using Opus → iOS listeners cannot play
  AAC → works on every device, every OS

Opus is used for:
```

```
  Voice calls (WebRTC — different transport)
  Voice notes (file-based, not streaming)

AAC is used for:
  VP Live audio track (in video stream)
  VP Audio Radio (HLS streaming)
  VP Audio Spaces listener HLS output

AAC at 32kbps for EA:
  Acceptable speech quality
  ~15MB per hour
  Works on any 2G connection
  Universal device support
```

---

6. VP Audio Spaces — Multi Speaker Rooms

The Concept

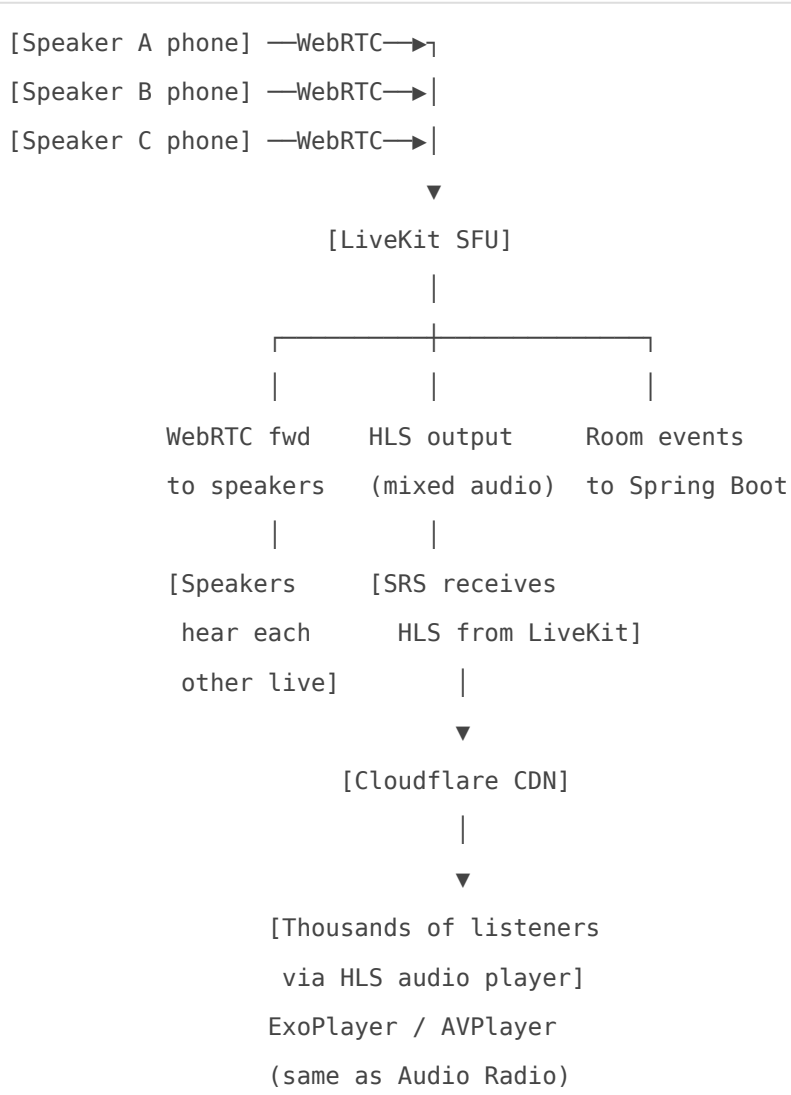
```
Not one broadcaster → many listeners
Multiple people in a shared audio room
Some speak, many listen
Listeners can raise their hand to speak
Host controls who gets the mic

Like Twitter Spaces, Clubhouse, Discord Stage Channels

Key insight:
  Speakers need LOW LATENCY (<200ms)
  to have a natural conversation
  HLS (6-15s delay) is too slow for speakers

  Listeners just need to HEAR clearly
  HLS delay is fine — they're not responding
  HLS scales to millions via CDN
```

```
Solution: TWO transport layers in one room
  Speakers    → WebRTC (LiveKit SFU) → <200ms
  Listeners   → HLS via CDN → 6-15s delay → millions scale
```

LiveKit SFU — What It Is

```
SFU = Selective Forwarding Unit

Traditional conference (MCU):
  Server mixes ALL audio into one stream
  Sends mixed stream to everyone
  High CPU (server does all mixing)
  Simple client

LiveKit SFU approach:
  Each speaker sends audio once to LiveKit
  LiveKit forwards each speaker's stream
    to all other speakers
  Speakers' apps mix locally (device CPU)
  Much lower server CPU
  Lower latency
  Better quality (no mixing artifacts)

For listeners:
  LiveKit outputs a mixed HLS stream
  Goes through SRS → Cloudflare CDN
  Listeners get one mixed audio stream
  Same HLS pattern as Audio Radio

Who built LiveKit:
  The same team that built Twitter Spaces
  Then open sourced it
  Actively maintained, Docker ready
  Official Android + iOS SDKs available
```

Full Architecture




```
Room events (raise hand, join, leave):
  LiveKit → Spring Boot via webhook
  Spring Boot → Ejabberd MUC → all participants
  Ejabberd MUC → Listeners also see events
                 (who joined as speaker etc)
```

Raise Hand Flow

```
Listener wants to speak:
     │ taps "Raise Hand" □□
     │ sends via Ejabberd WS to MUC room:
     │ { type: RAISE_HAND, roomId: "space-abc" }
     │
     ▼
Spring Boot:
  Records raise hand request
```

```
  Notifies host via Ejabberd WS:
    { type: HAND_RAISED, userId, displayName }
  Host sees list of raised hands in UI
     |
Host taps "Allow to speak" on a listener:
     |
     ▼
Spring Boot:
  Calls LiveKit API:
    Update participant permissions:
      canPublish: true   ← now allowed to send audio
  Generate new LiveKit token for this user
    (speaker token, not listener token)
  Send token to user via Ejabberd WS:
    { type: SPEAKER_PROMOTED, livekitToken: "..." }
     |
Former listener's app:
  Receives promotion event
  Stops HLS player (was listening at 15s delay)
  Connects WebRTC to LiveKit with speaker token
  Starts sending audio
  Now hears speakers at <200ms latency
  Other speakers hear them immediately
     |
Host can also:
  Lower someone's hand (dismiss)
  Mute a specific speaker
  Remove speaker (back to listener)
  End the space entirely
```

Speaker vs Listener — Connection Types

```
┌──────────────────────────────────────────────────────┐
│                   Audio Space Room                   │
│                                                      │
│  Speakers (up to ~20-30):                            │
│    Connected via WebRTC to LiveKit                   │
│    Send and receive audio streams                    │
│    Latency: <200ms (real conversation)               │
```

```
│    Connection: persistent WebRTC                    │
│                                                     │
│  Listeners (unlimited):                             │
│    Connected via HLS to Cloudflare CDN              │
│    Receive mixed audio only                         │
│    Latency: 6-15 seconds (fine — just listening)    │
│    Connection: HTTP requests every 2s               │
│    Scale: millions — CDN handles it                 │
│                                                     │
│  All participants:                                  │
│    Connected to Ejabberd MUC room                   │
│    Text chat, reactions, raise hand events          │
│    Room membership awareness                        │
└─────────────────────────────────────────────────────┘
```

LiveKit Token System

```
Spring Boot manages all LiveKit tokens
(LiveKit has official Java SDK)

Host token:
  canPublish: true
  canSubscribe: true
  roomAdmin: true
  → full control, can speak, manage

Speaker token:
  canPublish: true
  canSubscribe: true
  roomAdmin: false
  → can speak, cannot manage room

Listener token:
  canPublish: false      ← cannot send audio
  canSubscribe: true     ← can hear speakers
  roomAdmin: false
  → receive only

Token generation:
```

```
  GET /audio-spaces/{spaceId}/join
  Spring Boot checks:
    Is user the host? → host token
    Is user an approved speaker? → speaker token
    Otherwise → listener token (gets HLS URL instead)
```

LiveKit Docker Config

```
livekit:
  image: livekit/livekit-server:latest
  container_name: livekit
  restart: unless-stopped
  ports:
    - "7880:7880"      # HTTP API (Spring Boot calls here)
    - "7881:7881"      # WebRTC TCP
    - "7882:7882/udp"  # WebRTC UDP (primary)
    - "50000-60000:50000-60000/udp"  # ICE relay ports
  volumes:
    - ./livekit/livekit.yaml:/etc/livekit.yaml
  command: --config /etc/livekit.yaml
```

```
# livekit.yaml
port: 7880
rtc:
  tcp_port: 7881
  udp_port: 7882
  use_external_ip: true

redis:
  address: redis:6379    # reuses existing Redis 

turn:
  enabled: true
  domain: turn.nexgate.com
  tls_port: 5349
  credential: "${COTURN_SECRET}"   # reuses existing Coturn 

room:
  max_participants: 10000
```

```
  empty_timeout: 300
```

```
LiveKit reuses:
  Redis → already deployed □
  Coturn → already deployed for calls □
  No new infrastructure beyond LiveKit container itself
```

---

7. Live Chat — Ejabberd MUC

All three live modes (VP Live, Audio Radio, Audio Spaces) use Ejabberd MUC rooms for real-time text interaction.

Room Lifecycle

```
Stream / space starts:
     |
Spring Boot → Ejabberd REST API:
  POST /api/create_room
  {
    name:    "live-{streamId}",
    service: "conference.nexgate.com"
  }
  Room created: live-abc@conference.nexgate.com
     |
Broadcaster / host auto-joined as moderator
     |
Viewers / listeners join room as participants:
  App connects Ejabberd WS
  Sends MUC join stanza:
  <presence to="live-abc@conference.nexgate.com/Kibuti">
    <x xmlns="http://jabber.org/protocol/muc"/>
  </presence>
     |
Comments sent as MUC messages:
  <message to="live-abc@conference.nexgate.com"
           type="groupchat">
    <body>Mzuri sana! □□/body>
```

```
  </message>
     |
All room members receive instantly
No delay — Ejabberd MUC is real-time
     |
Stream / space ends:
Spring Boot → Ejabberd REST API:
  POST /api/destroy_room
  { name: "live-abc", service: "conference.nexgate.com" }
Room destroyed, members disconnected
```

Special Events in Live Chat

```
Beyond text comments, the MUC room carries:

Reactions (emoji bursts):
  { type: REACTION, emoji: "□□", userId, displayName }
  Client renders floating emoji animation

Gifts:
  { type: GIFT, giftId, giftName, amount, userId, displayName }
  Client renders gift animation
  Spring Boot processes payment separately

Raise hand (Audio Spaces only):
  { type: RAISE_HAND, userId, displayName }
  Host sees in management panel

Speaker promoted (Audio Spaces only):
  { type: SPEAKER_PROMOTED, userId, displayName }
  All participants see "Amina joined as speaker"

Viewer count updates:
  Broadcast every 30 seconds from Spring Boot
  { type: VIEWER_COUNT, count: 12453 }

Product card dropped by broadcaster:
  { type: PRODUCT_CARD, productId, name, price }
  Viewers tap → go to VP Shop product page
```

```
  Commerce during live □
```

Viewer / Listener Count

```
Two sources of truth:

1. Ejabberd MUC occupant count:
   GET ejabberd REST /api/get_room_occupants_count
   { room: "live-abc", host: "conference.nexgate.com" }
   → exact WebSocket-connected count

2. Redis counter (includes HLS-only listeners):
   INCR live:{streamId}:viewers  → on HLS playlist request
   DECR                          → on playlist stop / timeout
   More accurate for Audio Radio/Spaces
   where many listeners never connect WS

Display count = Redis counter (higher, more accurate)
Spring Boot broadcasts to MUC every 30 seconds
```

---

8. Stream Key System

Stream Key Design

```
Stream key = single-use authentication token
Broadcaster uses it to connect RTMP to SRS
SRS validates with Spring Boot before accepting stream

Format: random 32-character alphanumeric string
Example: nx_live_a1b2c3d4e5f6g7h8i9j0k1l2m3n4

Lifecycle:
  PENDING   → generated, not yet used
  LIVE      → broadcaster connected, stream active
  ENDED     → stream finished normally
  EXPIRED   → generated but never used (24h TTL)
```

```
  REVOKED   → manually stopped by admin

One active stream per user at a time
Attempting second stream → rejected by Spring Boot validation
```

SRS Webhooks to Spring Boot

```
SRS fires these events to Spring Boot:

on_publish   → broadcaster connected RTMP
  Spring Boot: validate key, update status LIVE,
               notify followers FCM,
               create Ejabberd MUC room,
               create LiveKit room (if audio space)

on_unpublish → broadcaster disconnected
  Spring Boot: update status ENDED,
               trigger File Thunder for VOD,
               destroy Ejabberd MUC room,
               log stream duration + peak viewers

on_play      → viewer started watching HLS
  Spring Boot: increment Redis viewer counter

on_stop      → viewer stopped watching
  Spring Boot: decrement Redis viewer counter
```

---

9. File Thunder Integration — VOD After Stream

What Happens After Stream Ends

```
Stream ends (broadcaster taps End / disconnects)
     |
SRS fires on_unpublish webhook
```

```
      |
Spring Boot:
  Update stream record: status ENDED
  Trigger File Thunder for VOD processing
  SRS has saved full recording as .mp4
      |
      ▼
Spring Boot → File Thunder:
  POST /api/v1/upload/request  (HMAC signed)
  {
    ownerId:   broadcasterId,
    domain:    POSTS,
    context:   LIVE_RECORDING,
    filename:  "stream_{streamId}.mp4",
    mimeType:  "video/mp4"
  }
  Returns: presigned MinIO PUT URL
      |
Spring Boot pulls recording from SRS
Uploads to MinIO via presigned URL
POST /api/v1/confirm { fileId }
      |
      ▼
File Thunder VideoWheel processes:
  HLS transcoding (all quality variants)
  Thumbnail extraction (best frame detection)
  Watermark: "@{broadcasterUsername}"
  NO outro — live recordings are long
  NO shortClip — full stream only
  Store in nexgate-public bucket
      |
      ▼
File Thunder fires webhook: media ready
Spring Boot:
  Creates VOD post on broadcaster's profile
  "Watch replay" button appears
  Appears in VP Feed for followers
  Stream record linked to VOD fileId
```

New File Thunder Contexts for Live

```
Existing contexts (unchanged):
  SOCIAL_VIDEO      regular video posts
  DM_ATTACHMENT     files sent in DMs
  DIGITAL_PRODUCT   digital goods in VP Shop
  ...

New contexts added for live:
  LIVE_RECORDING    full stream VOD
                    VideoWheel — no outro, no shortClip
                    always HLS, always long

  AUDIO_RECORDING   audio space / radio recording
                    AudioWheel processes
                    outputs: .m4a (AAC)
                    podcast episode on profile
                    waveform extracted (like voice notes)
```

nexgate-live MinIO Bucket

```
Existing buckets:
  nexgate-raw      temp uploads
  nexgate-public   social content
  nexgate-private  DMs and private files
  nexgate-digital  VP Shop digital products

New bucket:
  nexgate-live     live stream segments only

Why separate:
  SRS writes directly here (not via File Thunder)
  Short TTL segments — deleted after stream ends + VOD ready
  Different CDN caching rules (10s TTL vs 1 year for VOD)
  Different access pattern (SRS writes, CDN reads)
  Easy to monitor storage growth separately

Lifecycle:
  Stream starts  → SRS creates live/{streamKey}/ folder
```

```
  During stream  → .ts segments written every 2 seconds
  Stream ends    → Spring Boot schedules cleanup job
  VOD confirmed  → delete nexgate-live/{streamKey}/ folder
  Total life:    stream duration + ~1 hour buffer
```

---

10. Codecs & EA Network Strategy

VP Live Video Codecs

```
Broadcaster encoding (phone → SRS):
  Video: H.264 (hardware encoder — mandatory)
         Software H.264 too slow for real-time on phones
         H.264 hardware support: every phone since 2013
  Audio: AAC 128kbps (RTMP standard)
  Container: RTMP (streaming protocol)

SRS transcoding (server-side):
  Receives H.264 + AAC
  Transcodes to HLS quality ladder:

  Quality    Video bitrate   Audio    Resolution  EA target
  ──────────────────────────────────────────────────────
  1080p      3 Mbps          128k     1920×1080   WiFi only
  720p       1.5 Mbps        128k     1280×720    4G
  480p       600 kbps        64k      854×480     3G
  360p       300 kbps        48k      640×360     2G minimum
  ──────────────────────────────────────────────────────

  ExoPlayer/AVPlayer auto-selects based on network
```

VP Audio Codecs

```
Audio Radio (broadcaster → SRS):
  Codec: AAC 128kbps
  Container: RTMP audio only

Audio Radio (SRS → HLS):
```

```
  128kbps → WiFi/4G listeners
   64kbps → 3G listeners
   32kbps → 2G listeners  (15MB/hour — affordable)

Audio Spaces (speaker → LiveKit):
  Codec: Opus (WebRTC standard)
  Adaptive: 32-64kbps per speaker
  Echo cancellation: mandatory (multiple people)
  Noise suppression: mandatory (EA background noise)

Audio Spaces (LiveKit → HLS for listeners):
  LiveKit mixes speaker streams
  Outputs mixed audio → SRS → HLS
  Same AAC ladder as Audio Radio
  Listeners hear all speakers in one stream
```

Adaptive Streaming — EA Principle

```
The player always knows the network speed
because it measures how fast segments download

Segment download faster than playback → upgrade quality
Segment download slower than playback → downgrade quality

For a viewer in Dodoma on shaky 3G:
  Opens stream → starts at 360p (safe default)
  Network good → player tries 480p
  Stays stable → tries 720p
  Network drops → immediately back to 360p
  No rebuffering if switch is fast enough

Buffer strategy:
  Player buffers 3-4 segments ahead (6-8 seconds)
  Gives time to switch quality before buffer empties
  Viewer may notice brief quality dip — never a freeze

NexGate player config recommendation:
  Min buffer: 6 seconds
  Max buffer: 30 seconds
```

```
  Quality switch: aggressive downgrade, conservative upgrade
  → Prioritize uninterrupted playback over quality
  → EA networks fluctuate — better to be at 360p than buffering
```

---

11. Docker Deployment

Full docker-compose for Live Features

```
# SRS Media Server
srs:
  image: ossrs/srs:5
  container_name: srs
  restart: unless-stopped
  ports:
    - "1935:1935"    # RTMP ingest (broadcaster connects here)
    - "8080:8080"    # HTTP API + HLS output
    - "1985:1985"    # SRS management API
  volumes:
    - ./srs/srs.conf:/usr/local/srs/conf/srs.conf
    - ./srs/logs:/usr/local/srs/logs
    - ./srs/recordings:/usr/local/srs/objs/recordings
  depends_on:
    - chat-service
  networks:
    - nexgate-internal

# LiveKit SFU (Audio Spaces)
livekit:
  image: livekit/livekit-server:latest
  container_name: livekit
  restart: unless-stopped
  ports:
    - "7880:7880"
    - "7881:7881"
    - "7882:7882/udp"
    - "50000-60000:50000-60000/udp"
  volumes:
```

```
    - ./livekit/livekit.yaml:/etc/livekit.yaml
  command: --config /etc/livekit.yaml
  depends_on:
    - redis
  networks:
    - nexgate-internal
```

SRS Config Highlights

```
listen              1935;       # RTMP port
max_connections     1000;

vhost __defaultVhost__ {

  # Validate stream key with Spring Boot
  http_hooks {
    enabled         on;
    on_publish      http://chat-service:8082/internal/stream/validate;
    on_unpublish    http://chat-service:8082/internal/stream/ended;
    on_play         http://chat-service:8082/internal/stream/viewer-join;
    on_stop         http://chat-service:8082/internal/stream/viewer-leave;
  }

  # HLS output for viewers
  hls {
    enabled         on;
    hls_path        ./objs/nginx/html;
    hls_fragment    2;          # 2 second chunks
    hls_window      10;         # keep last 10 chunks in playlist
  }

  # FFmpeg transcoding to multiple qualities
  transcode {
    enabled         on;
    ffmpeg          /usr/bin/ffmpeg;

    engine 360p {
      enabled       on;
      vcodec        libx264;
```

```
      vbitrate      300;
      vfps          15;
      vwidth        640;
      vheight       360;
      acodec        aac;
      abitrate      48;
      output        rtmp://localhost:1935/live360p/{stream};
    }

    engine 720p {
      enabled       on;
      vcodec        libx264;
      vbitrate      1500;
      vfps          30;
      vwidth        1280;
      vheight       720;
      acodec        aac;
      abitrate      128;
      output        rtmp://localhost:1935/live720p/{stream};
    }
  }
}
```

Traefik — RTMP Does Not Go Through Traefik

```
Important: RTMP is TCP port 1935
Traefik handles HTTP/HTTPS only
RTMP port 1935 exposed directly on VPS

What Traefik does handle:
  stream.nexgate.com → SRS port 8080 (HLS output)
  TLS termination for HLS delivery

RTMP broadcaster connects:
  rtmp://stream.nexgate.com:1935/live/{key}
  No TLS on RTMP (RTMPS is complex, not needed for launch)

HLS viewers connect via Cloudflare CDN:
  https://cdn.nexgate.com/live/{key}/master.m3u8
```

```
  Cloudflare pulls from SRS port 8080
  Traefik handles TLS for this path
```

---

12. Database Schema

live_streams

```
live_streams
──────────────────────────────────────────────
stream_id         UUID          PK
broadcaster_id    UUID          FK → users
type              ENUM          VIDEO / AUDIO_RADIO / AUDIO_SPACE
title             TEXT
description       TEXT
cover_file_id     UUID          File Thunder fileId (stream thumbnail)
stream_key        TEXT          UNIQUE, used for RTMP auth
status            ENUM          PENDING / LIVE / ENDED / EXPIRED / REVOKED
started_at        TIMESTAMPTZ
ended_at          TIMESTAMPTZ
duration_seconds  INT
peak_viewers      INT
total_viewers     INT
muc_room_id       TEXT          Ejabberd MUC room name
vod_file_id       UUID          File Thunder fileId after processing
created_at        TIMESTAMPTZ
```

audio_spaces

```
audio_spaces
──────────────────────────────────────────────
space_id          UUID          PK
stream_id         UUID          FK → live_streams
livekit_room_id   TEXT          LiveKit room name
host_id           UUID          FK → users
title             TEXT
status            ENUM          SCHEDULED / LIVE / ENDED
```

```
max_speakers      INT           default 30
started_at        TIMESTAMPTZ
ended_at          TIMESTAMPTZ
```

audio_space_participants

```
audio_space_participants
──────────────────────────────────────────────
space_id          UUID          FK → audio_spaces
user_id           UUID
role              ENUM          HOST / SPEAKER / LISTENER
joined_at         TIMESTAMPTZ
left_at           TIMESTAMPTZ
hand_raised_at    TIMESTAMPTZ
promoted_at       TIMESTAMPTZ   when promoted from listener to speaker
promoted_by       UUID          host who approved
```

stream_viewer_stats

```
stream_viewer_stats
──────────────────────────────────────────────
stat_id           UUID          PK
stream_id         UUID          FK → live_streams
timestamp         TIMESTAMPTZ
viewer_count      INT
quality_360p_pct  DECIMAL       % of viewers on 360p
quality_720p_pct  DECIMAL       % of viewers on 720p
avg_watch_seconds INT
```

---

13. Scale Path

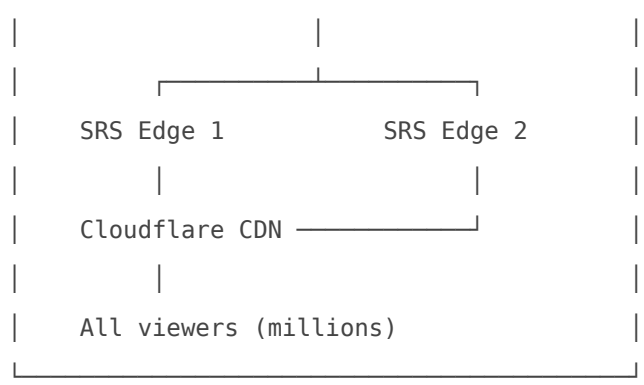
Current Architecture Limits

```
Single SRS node (Hetzner CPX31 — €19/month):
  Concurrent streams:  ~200 (with transcoding)
```

```
  Concurrent viewers:  ~50,000 (before CDN helps)
  Bandwidth:           20TB/month included

With Cloudflare CDN:
  Concurrent viewers:  Unlimited (CDN absorbs it)
  SRS only serves cache misses
  99%+ cache hit rate → SRS barely loaded

LiveKit single node:
  Concurrent spaces:   ~500
  Speakers per space:  up to 30
  Listeners per space: Unlimited (HLS via CDN)

This is enough for NexGate launch and
strong early growth — tens of thousands of users
```

Growth Stage — SRS Horizontal Scale

```
When 200 concurrent streams is not enough:

SRS Origin node:
  Receives RTMP from all broadcasters
  Passes stream to Transcode Farm

Transcode Farm (2-3 nodes):
  Each node handles FFmpeg transcoding
  Horizontal — add nodes as streams grow
  CPU-bound work distributed

SRS Edge nodes:
  Serve HLS to viewers
  Pull from Origin
  Multiple edges → load distributed

┌─────────────────────────────────────────┐
│  Broadcaster → SRS Origin               │
│                   │                     │
│              Transcode Farm             │
│              (3 nodes, FFmpeg)          │
```



WeChat EA Scale — Infrastructure

```
Broadcaster latency problem:
  Current: Broadcaster in Dar → stream goes to Hetzner Germany
           150-300ms upload latency
           Acceptable but not ideal

  At scale: SRS nodes in EA region
            Google Cloud Johannesburg
            OR AWS Cape Town
            Broadcaster → nearby SRS → low latency upload
            Better broadcaster experience

Storage cost at scale:
  Current: MinIO on Hetzner
  At scale: Cloudflare R2
            Zero egress cost (unlike AWS S3 which charges per GB)
            S3 compatible → zero code change to migrate
            At millions of viewer-hours: massive cost saving

Transcoding cost:
  CPU-heavy work
  At scale: GPU-accelerated FFmpeg nodes
            NVIDIA hardware encoding
            5-10x faster than CPU
            Lower cost per stream transcoded
```

Summary

VP Live and VP Audio Spaces are the live social layer of NexGate — living under VP Feed alongside regular posts, stories, and reels.

All three modes (VP Live video, VP Audio Radio, VP Audio Spaces) share the same infrastructure foundation. SRS handles all RTMP ingest and transcoding. Cloudflare CDN distributes HLS to unlimited viewers and listeners. Ejabberd MUC powers live chat and room events for all modes. File Thunder processes every stream into a VOD after it ends. Spring Boot manages stream keys, webhooks, room lifecycle, and all business logic.

VP Audio Spaces adds LiveKit SFU for the multi-speaker experience — speakers connect via WebRTC for real-time conversation while listeners receive the same HLS audio stream that Audio Radio uses, scaled to millions via Cloudflare CDN.

The EA network strategy is woven into every decision: HLS adaptive streaming down to 32kbps means VP Audio Radio works on 2G in rural Tanzania. Video quality ladders from 1080p to 360p ensure VP Live is accessible on 3G. The entire viewer and listener experience requires only ExoPlayer or AVPlayer — the simplest possible mobile integration.

For VOD, File Thunder's VideoWheel and new AudioWheel process every recording automatically after the stream ends — creating replay content with thumbnails, watermarks, and adaptive variants, stored in nexgate-public for CDN delivery. The live platform generates permanent content with zero extra work.

NexGate VP Live & VP Audio Spaces — Architecture v1.0 QBIT SPARK | SRS · LiveKit · HLS · Ejabberd MUC · File Thunder VOD

Revision #1
Created 13 July 2026 06:35:13 by Admin Qbit
Updated 13 July 2026 06:35:54 by Admin Qbit