Project Document: Centralized Video Playback System for Showroom TVs

Problem Statement :

Maruti Suzuki showrooms across multiple locations have TVs displaying promotional and informational videos.
Currently, there's no centralized system to manage what each TV plays.
->Our goal is to build a central hub that decides, distributes, and monitors video playback across all showroom TVs, regardless of brand or OS.

Quick requirements summary :

1. centralized control : The central hub should decide which video plays on each TV, using TV-specific Ids(TV static IP)
2. Offline Resilience : in the process we have to make sure that whenever a TV get disconnected to central hub or internet, video must continue to play, that means we have to store a video or two atleast in local storage of TV.
3. Health Monitoring : deploying a tracking mechanism in the TV to get live status of the TV weather video is playing or not, what's the storage status of TV.
4. Dashboard Visualization : An admin dashboard should show all TVs, their connectivity status, current video, and last-sync timestamp(whatever stats needed).

Our resources :

☑ **Videos** (named as `SR-ID_TV-ID.mp4` or similar).

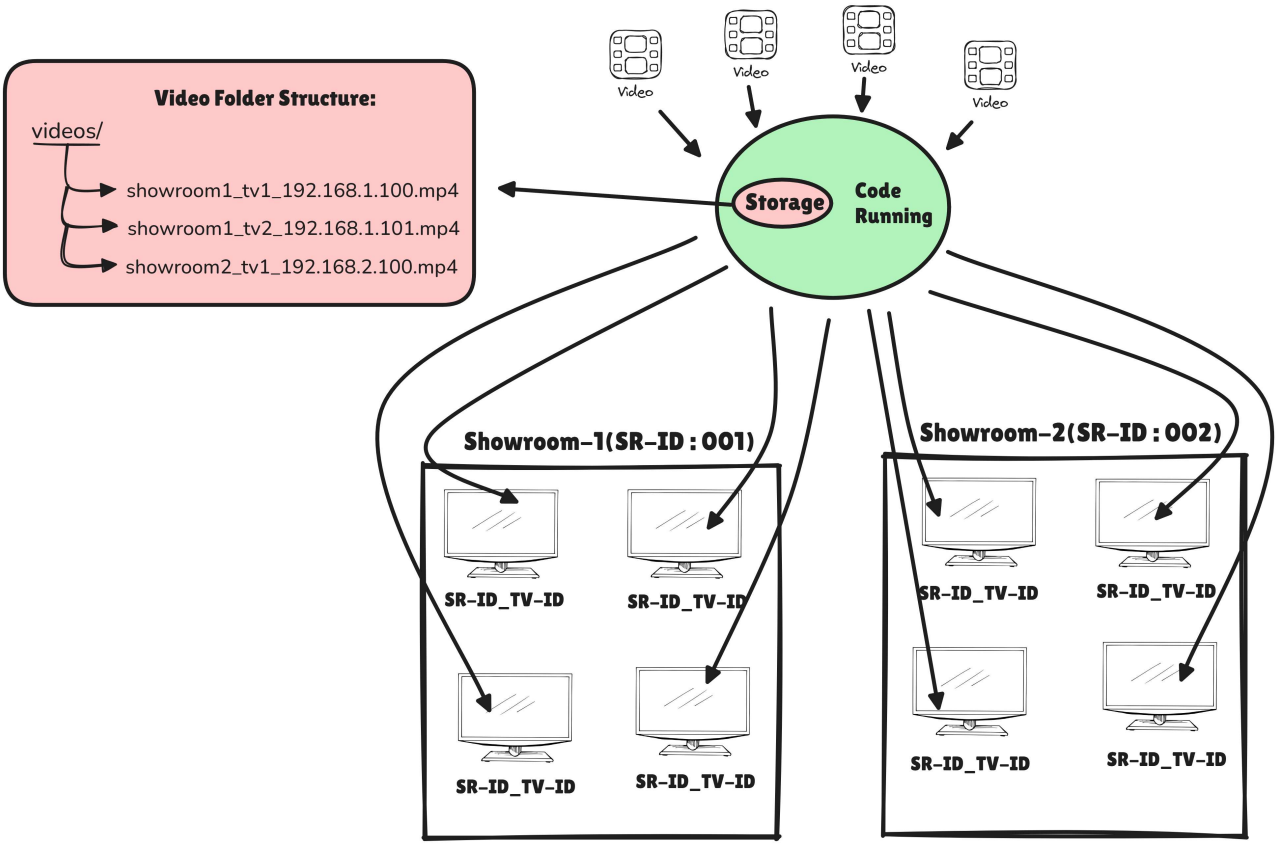
☑ **Central Hub** is basically a server — where videos are uploaded & control logic is running.

☑ **Static IPs & Credentials** for each showroom TV.

☑ **Raspberry Pi Modules** (confirmed by Mrityunjay Sir) connected to TVs via HDMI or USB.

- These can run Linux-based agents with Python.

System Architecture :



Central Hub Layer :

Video Management

- Hosts an upload folder (`/videos`), where new files are placed following a naming rule (`SR-ID_TV-ID.mp4`).
- Uses the **Python `watchdog`** library to detect when new videos appear.
- Maps each video to the correct showroom TV based on naming or a database table.

Content Distribution

- On detecting a new video, the hub automatically pushes it to the target TV/TV agent (Raspberry Pi / LibreELEC node) through:
    - POST method(HTTP POST) or MQTT topic publish or SFTP transfer
- The hub logs the dispatch status (success / failure / retry).

Tracking & Telemetry Reception

- Receives **heartbeat(here depicts hearbeat of video) and tracking data** from every TV/TV agent at regular intervals (e.g., every 30 seconds).
- Each report contains:
    - Current video name and timestamp position.
    - Playback state (playing / paused / stopped / error).
    - Free storage.
    - Last successful download and sync time.
- Data is stored in a monitoring database (PostgreSQL or MongoDB) for the dashboard.
- Alerts (email / SMS) can be triggered when a TV stops reporting or playback fails.

Dashboard API

- Exposes REST or WebSocket endpoints consumed by the dashboard frontend to show:
    - Online / offline status of all TVs.
    - Last reported video and health metrics.
    - Download / upload logs and errors.

below function is an excerpt from code at central hub. this function send video to TV give TV_IP and a port opened at TV .

```python
def send_video_to_tv(self, video_path, tv_ip):
    """Send video file to TV via HTTP POST"""
    try:
        url = f"http://{tv_ip}:8080/upload"

        logger.info(f"Sending {video_path} to TV at {tv_ip}")

        with open(video_path, 'rb') as video_file:
            files = {'video': video_file}
            data = {'filename': os.path.basename(video_path)}

            response = requests.post(
                url,
                files=files,
                data=data,
                timeout=300  # 5 minutes timeout for large files
            )

            if response.status_code == 200:
                logger.info(f"✓ Successfully sent video to {tv_ip}")
                return True
            else:
                logger.error(f"× Failed to send video to {tv_ip}: {response.status_code}")
                return False
```

Python's watchdog library :
Watchdog is a Python library and set of shell utilities designed to monitor file system events. It provides a cross-platform API to detect changes in files and directories, such as creation, modification, deletion, and movement.

we can employ this library for detecting the changes in the folder for example whenever a new videos get uploaded to folder in the central hub it will detect these changes and run the functions in the script responsible for sending the video forward to TV.

Communication Layer :

Establish a persistent connection between Hub ↔ TV Modules via one of:

1. getting remote access of TV using static IP and credentials.
And then we can run a executable file on the TV remotely which will install media player and tracking mechanisms.
Detailed Exploration on this approach : link

2. using raspberi pi.
We can connect raspberry pi to the TV via HDMI port and run a python script on the module that starts the port & sunsequently recieves the videos from central hub and play the videos, and send the tracking data. We can also add storage device to module if TV does not have enough storage for holding video. Main idea is that we interact with module and module executes actions in the TV.

Edge Layer (running script on TV side)

task in this layer is to run executable(python scrip) or lightweight program on the TV side for :

- Plays assigned videos automatically,
- Tracks playback status and local storage health,
- Reports this information back to the central hub.

This step is technically the most challenging, because it depends on the operating system and developer access level of each TV model.

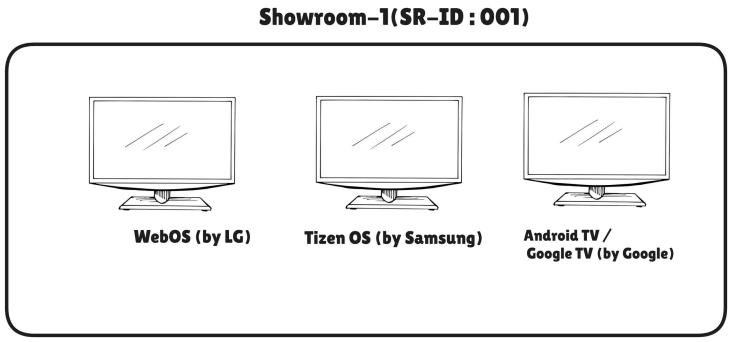
->Can We Run a Python Script Directly on TVs?

Short answer: No, not directly on most consumer Smart TVs.

Most Smart TV operating systems (LG webOS, Samsung Tizen, Roku OS, VIDAA, Panasonic My Home Screen, etc.) do not include a Python runtime environment and do not permit arbitrary code execution for security and stability reasons.
They only allow apps written using their official SDKs and app frameworks, which are sandboxed and distributed through their respective app stores.

Even though many of these systems are based on Linux kernels, they expose limited, proprietary user-space APIs—so we can't treat them as normal Linux machines that accept `python3 script.py`.

->Smart TV OS Landscape



Brand	OS	Based on	App Store
LG	**webOS**	Linux	LG Content Store
Samsung	**Tizen**	Linux	Smart Hub
Sony / TCL / Xiaomi	**Android TV / Google TV**	Android	Play Store
Amazon / Onida	**Fire TV OS**	Android fork	Amazon Appstore
TCL / Hisense	**Roku OS**	Linux	Roku Store
Hisense / Toshiba	**VIDAA OS**	Linux	VIDAA Store
Panasonic	**My Home Screen**	Firefox OS	Panasonic Market

most of TV OS are based on Linux Kernal except few which are based on android or firefox or else.

They each have their own app model:

- **LG webOS:** HTML5/JavaScript apps (Enact), webOS JS services.

- **Samsung Tizen:** Web apps (JS) or .NET; media via AVPlay APIs.
- **Roku OS:** BrightScript / SceneGraph only.
- **VIDAA / Panasonic:** HTML5/JS-style apps with restricted APIs.
- **Amazon Fire TV OS & Android TV / Google TV:** Android apps (Java/Kotlin; C++ via NDK). You can *sometimes* sideload things like Termux/Python on Android TV, but it's hacky, fragile, and not suitable for production.

Solutions :

Option 1 – Native App per TV Platform

We could build dedicated applications using each platform's official SDK:

- **webOS** → Enact (HTML5 + JavaScript)
- **Tizen** → Web App (JS) or .NET App
- **Android TV / Fire TV OS** → Android App (Java/Kotlin / C++ NDK)
- **Roku OS** → BrightScript
- **VIDAA / Panasonic** → Restricted HTML5 apps

Each app would handle:

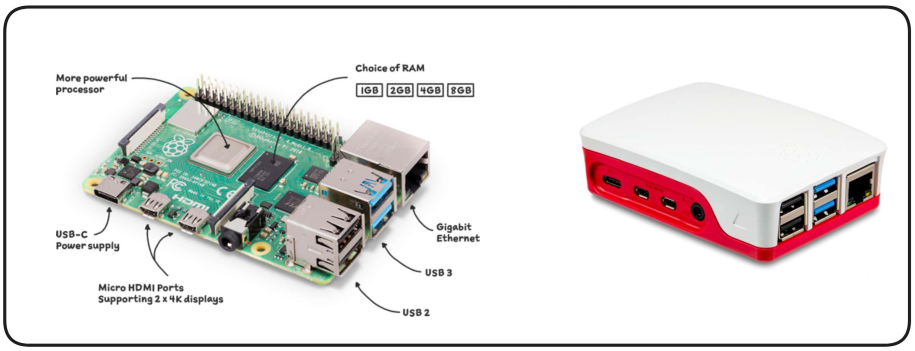
- Video playback via native media APIs,
- Local caching of media files,
- Periodic status reporting to the central hub.

However, this requires **multiple codebases**, **vendor certifications**, and **store publishing**, making it expensive and difficult to maintain at scale.

Option 2 – Using External Playback Device (Raspberry Pi + LibreELEC + Kodi + Python Add-on)

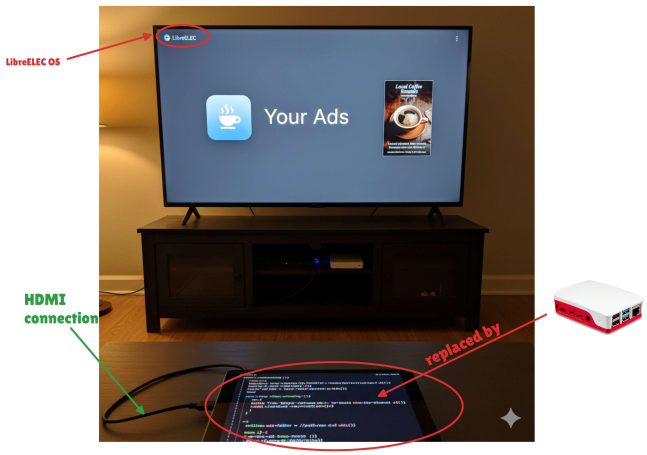
A more uniform and controllable solution is to use an **external device** connected via **HDMI**, such as:

- **Raspberry Pi / Intel NUC / Android Box**



WHAT is LibreELEC & Kodi : Understanding LibreELEC & Kodi

Example setup:



.....Step 1 – Hardware Connection

1. **Connect Raspberry Pi** to the TV via the **HDMI port**.
    - Power the Pi using a reliable 5 V / 3 A adapter.
    - Optionally connect a USB flash drive or micro-SD card (≥ 32 GB) for local caching.
2. Connect the Pi to the **internet** (Ethernet preferred / Wi-Fi optional).
3. Once powered on, the TV will detect HDMI input → select the Pi's HDMI source as default display.

.......Step 2 – Install LibreELEC

1. On a laptop, use the **LibreELEC USB-SD Creator** tool.
2. Flash the **LibreELEC OS image** onto the Pi's micro-SD card (model-specific build).
3. Insert the card into the Pi → boot.
4. On first boot:
    - Configure language, network, and hostname.
    - Enable **SSH** and **Samba (SMB)** in *Settings → LibreELEC → Services*.
      This allows remote file transfer and remote debugging later.

LibreELEC boots directly into **Kodi**, the media-center interface.

....Step 3 – Enable Kodi Remote & Developer Interfaces

1. In *Kodi → Settings → Services → Control*:
    - Enable **"Allow remote control via HTTP"** (default port 8080).
    - Enable **"Allow remote control from applications on other systems."**
2. Note the device IP address (e.g., `192.168.1.42`); the central hub will use this for API calls or data push.

....Step 4 – Add the Python Environment (Kodi Add-on)

Kodi 19+ ships with **Python 3** built in for add-ons.
Our playback-and-tracking logic will live inside a **custom Kodi Add-on** written in Python.

4.1 Add-on Folder Structure

```
plugin.video.showroomplayer/
│
├── addon.xml
├── icon.png
├── default.py          # Main entry point
├── resources/
│   └── settings.xml
└── service.py          # Background service
```

4.2 Key Files

- **`addon.xml`** – metadata & entry points.
- **`default.py`** – handles manual UI or playlist testing.
- **`service.py`** – auto-starts on boot, runs background loop.

Example `addon.xml` snippet:

```
<addon id="plugin.video.showroomplayer"
       version="1.0.0"
       name="Showroom Player"
       provider-name="YourTeam">
  <requires>
    <import addon="xbmc.python" version="3.0.0"/>
  </requires>
  <extension point="xbmc.service" library="service.py" start="startup"/>
</addon>
```

.....Step 5 – Core Functionalities Implemented in Python(these are Kodi add-ons)

5.1 Receiving and Updating Videos

- The add-on periodically **polls the central hub API** or subscribes to an **MQTT topic** like `videos/<SHOWROOM_ID>/<TV_ID>`.
- On receiving a new video URL or file metadata:
    1. Downloads it via `requests` or `urllib`.
    2. Saves to local folder `/storage/videos/`.
    3. Maintains a simple SQLite/JSON list of cached videos.
- Uses hashing or timestamps to avoid duplicate downloads.

5.2 Video Playback (using Kodi API)

Kodi exposes a JSON-RPC API and Python module `xbmc.Player`:

```
import xbmc, xbmcgui, xbmcaddon, json, requests, time

player = xbmc.Player()

def play_video(file_path):
    if not player.isPlaying():
        player.play(file_path)
```

- The add-on auto-plays the next video when the current one ends.
- The playlist loops indefinitely to satisfy “always-on” showroom playback.
- On reboot, Kodi auto-starts and the add-on resumes playback.

5.3 Offline Resilience

- Videos are cached locally (`/storage/videos`).
- If the hub is unreachable, the service reads the local playlist and continues looping until connectivity returns.

5.4 Tracking and Telemetry

The script periodically collects and sends data such as:

```
status = {
    "tv_id": TV_ID,
    "current_video": current_file,
    "position_sec": player.getTime(),
    "duration_sec": player.getTotalTime(),
    "is_playing": player.isPlaying(),
    "storage_free": psutil.disk_usage('/storage').free,
    "cpu_temp": get_cpu_temp()
}
requests.post(HUB_URL + "/telemetry", json=status)
```

- Frequency: every 30 – 60 seconds.
- If offline, logs locally and retries later.

5.5 Auto-Start and Recovery

- `service.py` runs automatically at Kodi startup (declared in `addon.xml`).
- Even after power loss, LibreELEC → Kodi → service.py sequence restarts playback automatically.

Step 6 – Remote Management and Monitoring

1. **Central Hub Dashboard** receives telemetry and displays:
    - Online/offline status, current video, free storage, last update time.

2. **Kodi Web API** (port 8080) can also accept remote JSON-RPC commands for manual control:
    - `/jsonrpc` POST → `{"method": "Player.Stop"}`
    - `/jsonrpc` POST → `{"method": "Player.Open", "params": {"file": "path"}}`
3. **SSH/Samba** provide maintenance access (logs, config edits).

Execution Flow Summary

Stage	Executed Where	Description
Hardware setup	On-site technician	Connect Pi → HDMI → TV; power on.
OS installation	Laptop → Pi SD card	Flash LibreELEC and boot.
Add-on deployment	Developer → via Samba/SSH	Copy `plugin.video.showroomplayer` folder to `/storage/.kodi/addons/`.
First boot	Kodi auto-launch	`service.py` starts, registers TV, requests playlist.
Normal operation	Pi (continuous)	Receives videos, plays them, caches offline, sends telemetry.
Monitoring	Central Hub	Displays status via dashboard & alerts.

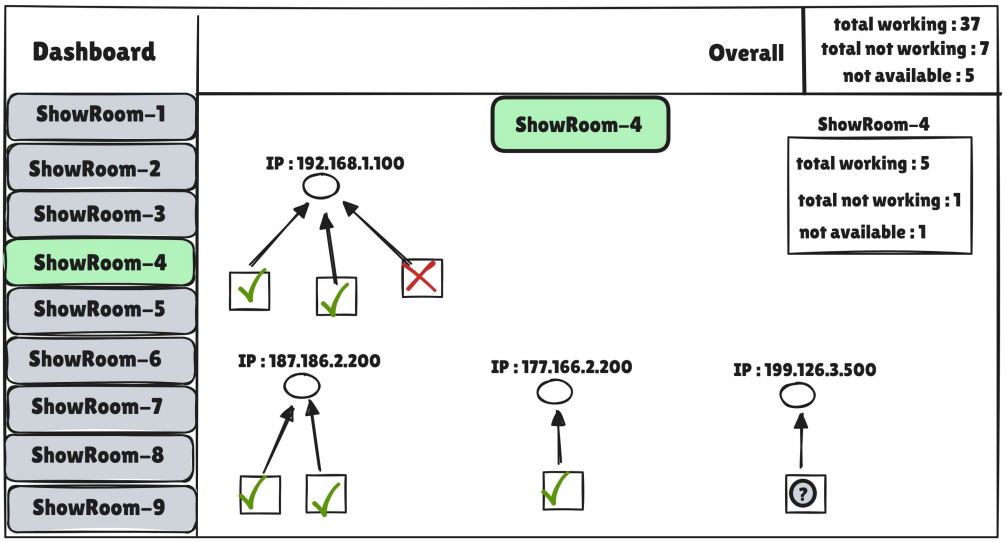
Why This Works

- **LibreELEC** provides a minimal, read-only OS → fast boot & stability.
- **Kodi** handles all video decoding and fullscreen display.
- **Python Add-on** gives complete automation and communication with the hub.
- **No dependence** on the TV's internal OS; works with any brand via HDMI.

Dashboard Design (High-Level)

below are a rough design how a dashboard might look like(obviously it will change as we progress in project building stage)

For Suzuki



For Us