ChatGPT

PRD: Low-Latency RTSP Video Streaming on Raspberry Pi CM5

Overview and Context

This product requirement document describes a **low-latency RTSP video streaming solution** for a Raspberry Pi Compute Module 5 (CM5) equipped with the official Raspberry Pi High Quality Camera. The system will run as a containerized service on a k3s cluster (lightweight Kubernetes) on the UAV's companion computer. The goal is to provide a real-time video feed from the UAV to remote clients (e.g. QGroundControl or ATAK) for manual drone piloting. **Latency is prioritized over image quality**, meaning the stream should be as close to real-time as possible, even if that requires reducing resolution or compression. The expected video quality is HD (at least 720p, up to 1080p), but it must be **configurable and potentially adjustable at runtime** to adapt to bandwidth constraints.

Key context: The UAV will announce the video stream via MAVLink to ground control stations (this announcement mechanism exists and is out of scope for this document). We assume the ground station will connect to the stream using an RTSP URL. The solution will use **Elixir and the Membrane multimedia framework** as the software stack, leveraging Membrane's capabilities for building real-time streaming pipelines in a maintainable way. Packaging the solution in an Alpine Linux-based container ensures easy deployment and consistency across vehicles.

Out of Scope: The base OS configuration of the Raspberry Pi host and general k3s cluster setup are already in place and not covered in detail here. However, any host adjustments specifically needed to access camera hardware or GPU features (for encoding) are considered in scope. The MAVLink integration and client-side setup (QGroundControl/ATAK configuration) are handled elsewhere and not described here beyond how the stream is exposed.

By focusing on a containerized Membrane-based design, we aim for a solution that is **maintainable, modular, and easily extensible** (e.g. to add features like recording or multi-stream support in the future). Below, we outline the specific requirements and the design choices made to meet them.

Requirements

Functional Requirements

- **Real-Time HD Video Stream:** Capture video from the Raspberry Pi HQ Camera and stream it in real-time with **low latency**. The system should support at least **HD resolution (720p@30fps)**, with a target of **1080p** if bandwidth and hardware allow. The resolution and frame rate should be **configurable** via startup parameters (environment variables or config file). For instance, one deployment might choose 1280×720 @ 60fps for lowest latency, whereas another could use 1920×1080 @ 30fps for higher detail. The default will be 1080p30, but this can be adjusted.

- **Low Latency Priority:** The end-to-end latency (camera capture to client display) should be minimized. **Sub-500ms latency** is a baseline goal, with stretch goals of ~200ms or better if possible. To achieve this, the pipeline must minimize buffering. We will configure the encoder for **"zero latency"** operation (no extra B-frames, low buffering) and ensure the RTSP/RTP stack doesn't introduce unnecessary delay. RTSP will be used only for session control; actual media will be transported via RTP/UDP for efficiency [1] [2]. On the client side (QGC/ATAK), documentation will recommend minimal buffering settings (e.g. in QGroundControl, using GStreamer's `latency=0` setting for RTSP sources).

- **RTSP Streaming Protocol:** The stream will be delivered over **RTSP (Real-Time Streaming Protocol)**, which is a standard for IP camera feeds. This provides compatibility with surveillance and UAV ground station software (QGC, ATAK) and allows multiple clients to request the stream on demand. RTSP will handle session setup (DESCRIBE/SETUP/PLAY) while the media is sent via RTP. Using RTSP aligns with the industry practice for IP cameras and ensures low-latency streaming [3] [4]. The RTSP server should listen on a configurable port (default 8554) and expose a stream path (e.g. `/video`).

- **Single Stream, Multiple Clients:** The system will produce one main video stream (from the single camera), but it **must allow at least two concurrent clients** to view the stream. In practice, this means one remote client (e.g. QGroundControl on a pilot's device) and one local client (e.g. a process in another container on the UAV for video analytics, or a local debugging viewer) can both receive the feed. The design should not assume a single consumer; it should either duplicate the stream to multiple sinks or utilize an RTSP server that can handle multiple subscribers to the same source. **At minimum, two simultaneous RTSP clients** should be supported with negligible performance degradation. This is important for future use cases like recording while streaming, or an onboard image processing module subscribing to the feed.

- **Configuration and Control:** It should be possible to configure key settings **at startup** (via environment variables or CLI args):

- Resolution (width x height) and framerate of the camera capture.
- Bitrate or quality level of the H.264 encoder (if using constant bitrate or VBR setting).
- RTSP endpoint details (port number, stream name, credentials if any).

Ideally, the design should also allow for **runtime adjustments** to quality if conditions change. For example, an operator might request a resolution drop from 1080p to 720p on the fly if the wireless link becomes weak. Fully dynamic, automated bitrate adaptation based on bandwidth (akin to adaptive bitrate streaming) is a stretch goal – if implementing that is too complex initially, a manual switch or requiring a restart with new settings is acceptable. The system architecture should be flexible enough that future updates can introduce dynamic adaptation without a complete redesign (for instance, by reconfiguring the pipeline or switching to a different encoding profile on the fly).

- **Hardware Video Encoding:** Leverage the **Raspberry Pi's hardware H.264 encoder** for video compression. Encoding must be offloaded to the GPU or dedicated video encoder block, rather than done in software on the CPU, to achieve full framerate and low latency with minimal CPU usage. The Raspberry Pi CM4/CM5's VideoCore GPU can handle 1080p30 (and even 1080p60 on newer models) H.264 encoding in real-time [5]. Using hardware acceleration ensures we meet performance requirements:

- The Membrane pipeline will utilize the **Membrane RPi Camera plugin**, which captures the camera feed and produces an H.264 byte stream encoded by the Pi's GPU [5] [6]. This means the camera frames are compressed to H.264 with minimal CPU overhead, directly on the Pi's video encoder. According to Raspberry Pi documentation, the encoding happens on the GPU, and even a modest Pi Zero could do 1080p30 via the hardware encoder [5].
- Hardware encoding not only reduces CPU load (freeing the CPU for flight control tasks) but also reduces latency by using the dedicated encoder pipeline which is optimized for real-time. We will configure the encoder for **baseline/low-latency profile** (no B-frames, possibly using an IDR keyframe every ~1 second or less to allow new clients to catch up quickly).

- **Note:** The Pi's H.264 encoder has known limits – e.g., maximum horizontal resolution of 1920 pixels for H.264 [7]. We will keep within these limits (1080p max). If future needs require higher resolution (e.g. 4K), H.265 encoding might be needed, but that is out of scope for now (and not all clients support H.265).

- **Live Streaming Only (for now):** The primary function is live streaming. There is **no requirement to record** the video onboard initially – we prioritize sending it live to the ground. However, the design should not preclude adding a recording feature later. In a future iteration, we may add the ability to record the stream to local storage (either the full resolution feed or a lower resolution copy) concurrently with streaming. This PRD will include a *Future Extensions* section describing how recording could be added (e.g. by teeing the stream to a file sink) so that the current design choices accommodate that possibility gracefully. For now, any file recording or cloud upload features are **excluded** from scope, keeping the system simpler and focused on real-time transmission.

- **Security and Access Control (optional):** Initially, the stream can be left unencrypted and open on the UAV's network (which is typically a closed private network between the drone and GCS). If needed, we should be able to **add basic authentication or access control** to the RTSP server (e.g. username/password or IP allowlist) to prevent unauthorized viewing. This is not a hard requirement for MVP, but the chosen RTSP solution should have hooks for this (for instance, many RTSP servers support credentials). We won't implement encryption (RTSP over TLS or SRTP) at this time due to added latency and complexity, but we note it for the future if operating in hostile environments.

Non-Functional Requirements

- **Maintainability:** The solution must be easy to maintain and extend by developers. By using the **Membrane Framework in Elixir**, we ensure the pipeline is constructed in a high-level language with good abstractions, rather than using complex shell scripts or low-level C/C++ code. Membrane's design aligns with Elixir's supervision trees and fault-tolerance, meaning the streaming pipeline can be monitored and restarted if issues occur, improving reliability. The choice of Membrane also means we benefit from an ecosystem of plugins for various formats and protocols (RTP, RTSP, MP4, etc.) [8]. Key maintainability considerations:
- Code clarity: The pipeline will be defined in Elixir, making the flow of data (source → encoder → payloader → network) explicit and easy to modify. This is preferable to opaque GStreamer command-line pipelines.
- Modularity: Each component (camera source, encoder, RTSP server, etc.) is a pluggable element. We can swap implementations (for example, switch to a different encoder or add a recording element) without rewriting everything.

- Community support: Membrane is an active open-source project, trusted for real-time multimedia [8]. We anticipate updates and community examples that we can leverage, keeping our solution up-to-date with minimal effort.

- Logging & metrics: We will integrate Membrane's logging and telemetry to trace pipeline performance. For example, we can log if frames are dropped or if encoding is lagging. In Kubernetes, the container logs and metrics can be collected for monitoring streaming health.

- **Performance:** The system must run efficiently on the Pi CM5. The container should use minimal CPU (thanks to hardware acceleration) and a reasonable amount of RAM. Real-time performance is crucial: the pipeline should consistently handle the target framerate (e.g. 30 or 60 FPS) without stalling. We will test the solution under load to ensure it can sustain ~30 Mbps video output (which might be needed for 1080p60 at low compression). **GPU memory** on the Pi must be sufficient – we will likely recommend setting `gpu_mem` **to 128 MB (or 256 MB for higher framerates)** on the host to accommodate the camera ISP and encoder buffers [9]. If the Pi is using the legacy camera stack, `start_x=1` (enable camera) must be set in `/boot/config.txt` [9], but on newer libcamera stack this is handled differently (still, ensuring the camera interface is enabled via device tree is required).

- **Reliability:** The stream should run continuously without manual intervention. If the camera or encoder fails or encounters an error, the system should attempt to recover (e.g. restart the pipeline or notify through logs). Membrane's supervision can be leveraged here – for instance, the Membrane pipeline can be linked to a supervisor that restarts it on crash, similar to how other Elixir processes are managed. In a k3s context, we can also have Kubernetes liveness probes. A simple approach is a periodic health check: the container could expose an HTTP or gRPC endpoint that indicates if the pipeline is running. At minimum, we'll use Kubernetes' restart policy such that if the process exits, the container restarts automatically.

- **Scalability (within context):** We are only dealing with one camera per UAV. Scaling here refers to possibly increasing the number of clients or adding additional cameras in the future:

- The design should allow adding a second camera stream in the future (e.g. a thermal camera). While not required now, a second pipeline could be instantiated for another camera module or video source. Membrane would support multiple pipelines or composite pipelines in one app.

- The RTSP server approach means adding more clients does linearly increase network and encoding load. We assume 2-3 clients max, which is fine. If many clients needed to view (e.g. 5+ viewers), we might then consider a different approach (like a ground station relay server). For now, 1-2 clients is the target.

- **Deployability:** The entire solution will be packaged as a Docker container based on **Alpine Linux** (chosen for its small footprint and compatibility). The container image should be as lightweight as possible, since many UAVs have limited storage and we want fast deployment. Alpine's package manager will be used to install any native dependencies (e.g. `libcamera` libraries, GPU drivers, etc.). Elixir and Membrane will be included (likely by building a minimal Elixir release). We need to ensure **all required OS libraries for camera and encoding are present** in the container:

- The **libcamera library** and associated tools (formerly *rpicam-apps*, like `libcamera-vid`) must be available inside the container because the Membrane RpiCam plugin relies on them [6]. We will either use Alpine packages (Alpine has `libcamera` and a `libcamera-raspberrypi` package [10]) or build them. The plugin may execute `libcamera-vid` under the hood to capture H264, so we need that binary. If Alpine's `libcamera-raspberrypi` doesn't include the apps, we will compile the **libcamera-apps** from source as part of the Docker build (there are community references for doing this on Alpine [11]).
- GPU drivers: The container must have access to the Pi's GPU for encoding. On Raspberry Pi, this typically means giving access to `/dev/video*` devices (which include the camera device and possibly the video encoder device) and potentially `/dev/vchiq` (the GPU communication interface). In k3s, we'll configure the Pod with the necessary device mounts or run it in privileged mode if needed. We will document that the deployment manifest should include something like:
    - `devices: ["/dev/video0", "/dev/video1", "/dev/video10", "/dev/video11", "/dev/vchiq"]` (these are examples – on libcamera, `/dev/video0` is the camera, and the encoder might be accessed via V4L2 M2M on another device, or via libcamera's API internally).
    - Alternatively, use `privileged: true` for simplicity, with an understanding of the security trade-off.

- The container should run as root (since it needs low-level hardware access), but within the pod's security constraints.

- **Alpine Base Image:** We explicitly choose Alpine for the base to keep the container small and secure. Alpine is a minimal distro that still has packages for media libraries we need (ffmpeg, gstreamer, or libcamera). Our build pipeline will produce an optimized release (using mix release) so that we only include necessary BEAM files and Membrane plugins, trimming any build tools from the final image. We anticipate the final image to be on the order of tens of MBs (excluding perhaps some weight from libcamera/ffmpeg libraries). Using Alpine also aligns with many Kubernetes setups where lightweight images are preferred.

- **Standards and Compatibility:** We stick to standard protocols and formats:

- Video codec: **H.264** (Baseline or Main profile) which is widely supported by clients and provides hardware acceleration on both Pi (for encoding) and most client devices (for decoding). This ensures QGroundControl, ATAK, or VLC can easily play the stream. (H.265/HEVC could reduce bandwidth, but it introduces decoding latency and is not as universally supported; we choose H.264 for now).

- Container/stream format: **RTP over RTSP**, following RFC 2326. The stream's SDP will be communicated in the RTSP setup, including the H264 codec parameters (SPS/PPS). We will ensure the stream is configured to periodically send **SPS/PPS (sequence headers)**, so a client that joins can decode without waiting for a keyframe. For example, the H264 RTP payloader can be set to send config data (in some systems this is `config-interval=1` every second) – Membrane's RTP H264 payloader will handle this, as it conforms to the RTP payload spec for H264. This is important for smooth client experience when connecting mid-stream.

- **Future-Proofing:** Design decisions should consider the future where:

- Recording can be added.
- Quality adaptation might be automatic.
- Possibly switching to a different protocol (WebRTC or SRT) if needed for lower latency or firewall traversal. The current architecture using Membrane will allow us to swap out or add modules (Membrane has WebRTC and SRT plugins available) [12] [13]. For now RTSP is the requirement, but the investment in Membrane means we're not stuck with RTSP forever – we could leverage the same captured feed and send it via multiple protocols if needed.

In summary, the requirements ensure the UAV video stream is **fast, configurable, and robust**, and that the implementation is clean and maintainable for developers.

System Design and Architecture

High-Level Architecture

The streaming solution is built as a **Membrane multimedia pipeline** running inside an Elixir application (OTP release). The pipeline captures video from the camera, encodes it, and serves it over the network via RTSP. Below is a high-level block diagram of the components involved in the pipeline and how they interact:

Figure: Membrane Pipeline for RTSP Streaming on Raspberry Pi. The camera feed (CSI camera) is captured by the Membrane RpiCam Source element, producing an H264 byte stream. This is passed through a payloader to packetize it into RTP. The RTP stream is then served to clients via an RTSP server component. Optionally, a Tee can duplicate the stream for local processing or recording. [6] [14]

Explanation of Components:

- **Membrane.RpiCam Source:** At the start of the pipeline is the Membrane RpiCam plugin's source element. This element interfaces with the Raspberry Pi camera module through the libcamera stack [6]. It abstracts the complexity of camera drivers by spawning the `rpicam-app` (libcamera-vid) under the hood to get an H.264 stream. The source is configured with desired `width`, `height`, and `framerate` (or left as `:camera_default` to use the camera's default mode) [15] [16]. It outputs a bytestream of H264 data (aligned to frame boundaries, with SPS/PPS included as needed) [17]. Essentially, this element takes the place of using `raspivid` or `libcamera-vid` manually – it uses the GPU to encode video and directly provides that stream to the pipeline. Using this plugin means we don't manually handle raw frames or call ffmpeg; it's a clean integration within Elixir.

- *Rationale:* Using Membrane's RpiCam source encapsulates all hardware-specific details. It ensures that if the camera is not ready or fails, we get an error in Elixir we can handle. It also avoids needing a separate process for the camera – Membrane runs it and can stop it cleanly. This improves maintainability and reliability (no orphan `libcamera-vid` processes, etc.). According to the plugin docs, one can specify a `:timeout` as well (which defaults to infinity) to automatically stop after some duration for testing [18], but in our case we will run indefinitely.

- **Membrane H264 Payloader (RTP):** The H264 bitstream from the camera source is next fed into an RTP payloader element (`Membrane.RTP.H264.Payloader`). This component takes raw H264 frames and packetizes them into RTP packets according to the RTP/H264 format (FU-A fragmentation for large NAL units, etc.). It adds the proper RTP headers, timestamps, and markers. Membrane has a ready plugin for H264 RTP payload [19] [20] which we will use. We will configure it to emit periodic keyframe parameter sets if needed (to assist client decoding). Typically, the payloader works with a jitter buffer on the receiver side, but for our server, it will just push out packets. The payloader will be connected to a network sink.

- **RTSP Network Sink/Server:** This part is responsible for actually sending the RTP packets to any connected clients via RTSP. There are a couple of design choices here:

- **Membrane-based RTSP server**: We intend to use Membrane's capabilities to implement an RTSP server within our application. Membrane doesn't yet have a fully high-level "RTSP Server" element (as of the latest version) that accepts a live pipeline, but we have low-level components to assemble one. We will leverage the `membrane_rtsp` package (which provides RTSP session handling as a client/server library) and the `membrane_rtsp_server` (an experimental simple server) to handle RTSP commands [21] [22]. In practice, we can create a small GenServer that listens on the RTSP TCP port, handles RTSP DESCRIBE/SETUP/PLAY requests (using Membrane's SDP and RTSP utilities to respond with the correct SDP containing our stream's parameters), and then link the client's request to our pipeline's output. Each client will get an RTP stream (over UDP or interleaved TCP as needed). Membrane's RTP and UDP plugins will be used to send the packets out [23] [24]. We will ensure the pipeline remains running continuously, sending packets only when there are clients (to save bandwidth). If no clients are connected, we may choose to drop the outgoing data (or not start the source until a client connects, to save CPU – although the overhead of encoding when not used might waste some power, so perhaps we can start/stop on demand).

- **Alternate approach (External RTSP server)**: As a fallback or interim solution, we considered using an external RTSP server like *MediaMTX (rtsp-simple-server)* [25] [26]. In this design, our Membrane pipeline would push the stream to the external server which would handle client connections. For example, Membrane could send RTP to the local server via a UDP or a RTSP push. The external server is well-tested and easily supports multiple clients, authentication, etc. However, adding another component (with separate config) adds complexity and slightly reduces maintainability (two different systems to monitor). **Our preference is to implement the RTSP serving within the Membrane app** for a more integrated solution. Membrane's own "Simple RTSP Server" example is currently limited to serving a file [27] [28], but we can adapt the concept to serve live data by replacing the file source with our camera pipeline.

In summary, the application will effectively act as an **RTSP IP camera server**. A client connecting will perform the standard RTSP handshake: we'll advertise an SDP with a video track (H264 codec, RTP payload type 96, and includes the codec initialization data). Once the client sends RTSP PLAY, our pipeline's RTP stream will flow. If multiple clients connect, each will go through SETUP and get their own RTP session (we can use the same encoded stream bits for all to avoid re-encoding). The Membrane framework, combined with Elixir's concurrency, should allow handling a few clients easily.

- **Tee (for Multi-Output)**: To support multiple simultaneous outputs from one source, we may include a **Membrane Tee element** in the pipeline [14]. The Tee takes one input pad (the H264 stream) and replicates it to multiple output pads. This could be used in two ways:
- Feeding multiple RTP sinks (for multiple clients) if we needed to manually manage different network outputs. However, if the RTSP server handles duplication internally (likely it will send the same packets to all subscribed clients), we might not need an explicit Tee for that.

- Enabling **future extensions** like local recording or local processing. For example, one branch of the Tee could go to a File sink (to record an MP4), while the other branch goes to the RTP/RTSP sink for live streaming. For now, we may not activate the Tee (just have a single branch), but by designing the pipeline with this in mind, we ensure that adding a second branch later (which Membrane supports dynamically or by pipeline re-init) is straightforward. Membrane's plugin library includes a `membrane_tee_plugin` specifically for splitting streams [29], indicating this is a standard practice.

- **Pipeline Supervision and Control:** The Membrane pipeline will run under a supervisor within the Elixir application. This means if the pipeline crashes (e.g., camera disconnect, or an unexpected error in encoding), it can be automatically restarted without bringing down the whole app. The restarts can be throttled to avoid rapid loops (e.g., if camera is physically unplugged, it might keep failing). We will also provide an API or CLI for the application if we need to trigger reconfiguration. For instance, an admin could `rpc` into the Elixir app to change resolution at runtime: implementing this might involve stopping the current pipeline and starting a new one with different source settings. Membrane allows dynamic pipeline modifications, but sometimes it's simpler to reconstruct the pipeline. We can design it such that a configuration change event (perhaps coming from a Kubernetes config update or a MAVLink command via another service) triggers a pipeline restart with new parameters. The downtime for a restart would be on the order of a second or two, which might be acceptable for a deliberate quality switch.

- **Networking considerations:** By default, RTSP will use UDP for RTP data. We need to ensure the k3s setup allows UDP traffic on the chosen port (8554 or custom) from the drone to the ground control. Likely, the ground control device will be on the same network via a telemetry radio or Wi-Fi. If NAT or firewalls are an issue, we might consider RTSP-over-TCP (interleaved mode) as a fallback; Membrane's RTSP handling can detect if the client requests RTP/AVP/TCP and then send the stream interleaved in the RTSP TCP connection. This is beneficial if only a single TCP port can be used. We will support both UDP and TCP transport for RTP to be robust:

- QGroundControl typically works with UDP, but ATAK might sometimes prefer TCP. Our server should allow either as per RTSP standard.

- We will test the stream in both modes to check latency (UDP is expected to be slightly lower latency and less taxing on the CPU due to no retransmits, so that's the primary path).

- **Memory and Buffering:** We will carefully tune buffering in each element for low latency. For example, the camera source might have an internal queue – we will use the smallest feasible buffers. The RTP payloader and network sink will also use minimal buffering (just enough to smooth over network jitter). If Membrane's default is more oriented to throughput, we might adjust it. The target is that each frame is sent out almost immediately after capture and encode, not piling up anywhere. **RTCP** (Real-Time Control Protocol) might be utilized by the RTSP server to monitor quality – e.g., the server could listen for receiver reports about packet loss. In future, this could inform adaptive bitrate logic. For now, we may not actively use RTCP feedback, but the plumbing will be there as part of RTP sessions.

Hardware Utilization & Host Setup

To meet the performance goals, we exploit the Raspberry Pi's hardware capabilities fully: - The **Camera ISP and sensor** will be driven by libcamera. The HQ Camera (12MP Sony IMX477) can output various modes; for 1080p30 it will likely use a binned or cropped mode. We'll choose a mode that supports our desired resolution and frame rate with the lowest latency (likely the video port of the camera). - The **VideoCore VI (on Pi 4/5)** provides the H.264 encoder. We need to ensure it has enough contiguous memory. As mentioned, setting `gpu_mem=128M` (or 256M if 4K frames were ever attempted) is recommended [9]. On newer OS (Bullseye/Bookworm with libcamera), the memory split is still relevant for the firmware encoder. We will include this note in deployment docs. - The container must have access to `/dev/video0` (camera). We will use Kubernetes device plugins or a privileged container to allow this. If using a device plugin, we might label the node and use tolerations to schedule this pod only on the Pi with the camera. - The **thermal and CPU load** on the Pi should be monitored. Encoding 1080p60 can heat up the Pi; we might cap to 1080p30 if we find temperature is an issue. The design is such that the heavy lifting is on the GPU, which can handle it, but we will still test under continuous use in expected ambient temperatures to ensure no throttling occurs. If needed, we'll suggest enabling the Pi's cooling (e.g., a fan or heatsink on the CM5 carrier). - No significant host software dependencies are needed beyond what's in the container. We do rely on the kernel drivers (V4L2, media controller, etc.) being present and the camera being enabled. It's assumed the host OS (likely Raspberry Pi OS Lite or a Yocto-based system for CM5) is already configured with the camera drivers active. We will document that requirement: e.g., on Raspberry Pi OS, run `raspi-config` to enable Camera Interface and I2C if required (HQ cam uses CSI-2 and I2C for control).

Maintainability & Design Choices

Throughout the design, choices were made to maximize maintainability: - **Elixir/Membrane Framework:** Using Membrane in Elixir aligns with the rest of our software stack and provides a high-level, declarative way to build the streaming logic. This reduces the "glue code" required and allows us to tap into Membrane's existing plugins (for camera, for RTP/RTSP, file sinks, etc.). Membrane pipelines are supervised processes, which fits naturally into an Elixir application structure [30]. This means if a part of the pipeline crashes (say the camera element encounters an error), it doesn't crash the whole system uncontrollably; we can catch it and handle it (maybe by restarting or by emitting an event that operators can see). - **Plugin Reuse vs Custom Code:** Wherever possible, we prefer using well-tested components over custom implementations: - *Camera integration:* The Membrane RpiCam plugin is used instead of writing a custom NIF or calling `ffmpeg` in a port. This plugin is maintained by the Membrane team (Software Mansion) and is specifically made for Raspberry Pi cameras [6], which gives us confidence in its reliability and performance. - *RTSP server:* If the Membrane's own RTSP server pieces prove insufficient or too raw, we are open to using a minimal external server (like wrapping *rtsp-simple-server*). That server (MediaMTX) is a single binary ~10MB that's easy to include, and it's written in Go with a focus on simplicity. It could run in the same container and we feed it via a FIFO or UDP. However, this introduces a second system to configure (need to supply it a config file, manage its process, etc.). For maintainability, our first attempt will be making the Elixir app serve RTSP natively so everything is in one place. We note that Membrane's simple RTSP server was primarily for streaming an MP4 file [27], so serving a live stream might require development. We're prepared to allocate some effort to extend or adapt it for live pipelines – this investment is worthwhile long-term, as it keeps our stack unified. - **Configurability:** We will implement reading configuration from environment variables in the container. For example: - `STREAM_RESOLUTION=1280x720` and `STREAM_FPS=60` might be env vars. The Elixir app will parse these and pass to the RpiCam source options (width=1280, height=720, framerate= {60,1} assuming it expects a fraction). - `STREAM_BITRATE=4000000` (in bits, e.g. ~4 Mbps) could be used if we had control over encoder bitrate. The Membrane RpiCam plugin might allow setting the quality or we might need to adjust `profile` or `level`. If such options aren't exposed, we might rely on the camera's default rate control (which often is constant quality). - `RTSP_PORT=8554` and `RTSP_PATH=/video` could set where the server listens.

Having these in the env means we don't need to rebuild the image for different drones or different network conditions – it's configured at deployment time (in a ConfigMap or deployment manifest). This is a crucial maintainability feature because it allows one codebase/container to serve multiple use-cases by configuration.

- **Monitoring and Logging:** The application will produce logs at different levels. We will include logs for:
- Pipeline initialization (print out chosen resolution, framerate, etc.).
- RTSP connections (log when a client connects/disconnects, and who, for audit).
- Errors (if the camera fails or if a frame is dropped due to overload, etc.). Membrane has an internal logger we can enable for debugging pipeline issues [31] [32]. We'll likely run in an info or warning logging level normally to avoid too much output, but be able to turn on debug for troubleshooting.

- We might integrate a lightweight metrics solution: Membrane's telemetry events could be forwarded to an Prometheus endpoint or simply logged. E.g., we could log stats like "X fps being encoded", "network queue latency = Y ms" if available. This will help in optimizing and verifying performance.

- **Development and Testing:** To ensure the design works as expected, we will set up testing scenarios:

- Lab test with a Raspberry Pi (the same model as on the UAV) connected to a network with a laptop running QGroundControl. We will measure the latency (point a stopwatch in front of camera and compare video vs real). We will also test different resolutions and stress test two clients (e.g., QGC and VLC playing simultaneously).
- We will test what happens if a client disconnects and reconnects repeatedly (to ensure no resource leaks or crashes in the RTSP handling).
- Test low-bandwidth scenarios: artificially throttle network to see if the pipeline can handle it (e.g., do we see increased latency or frame drops?). This is to simulate the drone flying far and losing bandwidth. We might not implement adaptive response now, but we want to ensure the system doesn't crash – it should ideally just show a degraded video or freezing frames on the client, and recover when bandwidth improves.

- Edge cases: camera unplugged or not found (should log an error clearly), trying to set a resolution not supported by the camera (the plugin likely returns an error in init; we should handle and default to something safe).

- **Example Configuration:** By default, we can define a profile for the stream: 1080p30, bitrate ~4 Mbps, using baseline profile (for broad compatibility). If low latency is absolutely critical, we might use a lower resolution like 720p60, which has more frequent frames (thus less motion blur and less time between frames for control feedback). We will provide in documentation how to adjust these to the user's needs. Over time, with testing, we might determine a best trade-off (perhaps 720p60 yields better control-ability of the drone with ~150ms latency, whereas 1080p30 might be ~250ms and more taxing on bandwidth).

Future Extension: Recording Feature

Although out of scope for the initial implementation, it is useful to outline how we can extend this design to support **onboard recording** of the video stream, as this influences our current design decisions:

- **Concurrent Recording:** Using Membrane's graph structure, the simplest way to add recording later is to **add a branch to the pipeline** that writes to a file. We could insert a `Membrane.File.Sink` or `Membrane.MP4.Muxer` + `File.Sink` in parallel with the RTP streaming. For example, after the H264 encoder (or directly tapping the H264 bytestream), we use a Tee element [14] to split: one path goes to the RTP payloader (as now), another path goes to an MP4 muxer. Membrane has an MP4 plugin for packaging H264 into `.mp4` files [33]. We would likely also include an AAC audio track if audio was present, but in our case it's video-only so a simple MP4 with video track is fine.
- **Recording Control:** We might not want to record 100% of the time (to save storage). So, the pipeline might have the recording branch initially inactive. When a record command is issued (could be via MAVLink command or a ground station UI), we dynamically add and link the MP4 sink branch and start writing. Membrane supports adding new elements to a running pipeline (though it's advanced, it can be done with proper synchronization). Alternatively, we could always have the recording branch there but have it writing to a temporary buffer and not saving until triggered (less ideal).
- **File Management:** We'd need to handle file naming, rotation, etc. Possibly integrate with the UAV's logging system. This is designable later; for now, just ensure we can get the data. The important part is that by using Membrane, adding this feature doesn't require a separate encoding pass – we can reuse the same encoded frames for both streaming and recording, which is efficient.
- **Impact on current design:** The possibility of recording means:
- We keep the stream as H264 (which we are doing) since that's easily recordable. If we were doing raw video to stream, recording would then need encoding; but we are already encoding, so recording is just saving bytes.
- The Tee should be planned (which we have). If not implementing now, at least leave a clean point in code where it can be added.
- Ensure the H264 stream is compatible with MP4: e.g., include SPS/PPS in-band or provide them to the MP4 muxer. This typically means the first few frames have the SPS/PPS which is standard. Membrane's MP4 muxer likely will handle H264 stream format if provided with codec info (like resolution, avcC config). We might need to feed it the stream format info from the camera source (Membrane elements usually exchange caps that include such codec details).
- The container must have enough storage I/O performance to write the video. On a CM5, usually there's an eMMC or SD card – writing a 4Mbps stream to disk is not an issue, but extended recording could fill storage. That's a policy matter (maybe only record on command or when important). Not directly affecting design, but we keep it in mind.

Dynamic Quality Adjustment

Another future feature is dynamic quality scaling. We mention it here to ensure the current design can support it: - If we detect low bandwidth (via RTCP or other telemetry), we might want to reduce bitrate. With H264, we could do this by instructing the encoder to lower quality (increase QP) or drop resolution. Some encoders allow changing bitrate on the fly. We need to investigate if libcamera-vid or the MMAL encoder allows dynamic reconfig. If not, we might have to stop and restart the camera with new settings (which causes a brief pause). - **Design impact:** We will design the system such that a "quality change" can be triggered through a single function that reinitializes the pipeline to new settings. This function would be the same as what reads the env config, but now using a new config at runtime. This way, whether the change comes from an environment update or an internal trigger, the code path is similar. Utilizing Elixir's hot-code swap isn't necessary; a simple restart of the pipeline process suffices. - Alternatively, one could prepare two parallel pipelines (hi and lo res) and switch which one is feeding the RTSP based on conditions, but that doubles resource use. The multi-stream approach used by others (like the blog which created two feeds HQ and LQ) [34] [35] is something we might consider in the future if adaptive streaming is critical. The Pi's camera can support multiple output streams from the sensor (using multiple splitter ports) [36], as demonstrated with one high-res and one low-res simultaneous stream. Our current design doesn't capture two streams, but knowing this is possible, we keep it as an option. Membrane RpiCam plugin might not natively expose multiple ports, so that might require running two RpiCam sources (if the underlying camera driver and libcamera support concurrent streams). This is a deep future exploration, but the key is our architecture (with Membrane) would allow adding another pipeline if needed. - For now, **manual dynamic** adjustment is the plan: operators can decide and send a command to reconfigure. Automated adaptation will be explored once we have real-world data on how often bandwidth issues occur and how quickly we need to react.

Implementation Plan

Below is a phased plan for implementing the solution, aligned with the requirements and design:

1. **Prototype Pipeline on Raspberry Pi:** Start by writing a simple Membrane pipeline on a Raspberry Pi (outside of Kubernetes). Use the Membrane RpiCam source to capture 10 seconds of video and save to a file to verify that the camera and encoder work [6] [37]. This validates hardware encoding and our ability to interface with the camera. Then extend the pipeline to stream via RTP to a known host (skip RTSP initially). For example, use Membrane UDP sink to send RTP packets to a laptop and play them with a GStreamer pipeline or ffplay to verify low latency. This step ensures the core video capture and RTP output are functional.

2. **Integrate RTSP Session Handling:** Implement or incorporate an RTSP server. This likely involves using Membrane's `membrane_rtsp` for parsing requests and `membrane_rtp_plugin` for setting up the RTP session [38] [39]. We will create an Elixir GenServer (call it `VideoStreamServer`) that on start binds to the RTSP port and listens for connections. On a DESCRIBE request, it responds with the SDP containing the media stream description (we can get the SPS/PPS from the first buffer or from the Membrane caps). On SETUP, it will prepare to send RTP to the client's indicated ports. We'll likely use UDP sockets via Membrane's UDP plugin bound to a port (or even use the TCP connection for interleaved). The Membrane pipeline can be started at system boot and just idly encoding frames; or we could start encoding on the first PLAY request. A simpler approach is to start encoding at boot (so the latency from request to first frame is minimal) and just drop packets until a client connects. We will decide based on power usage considerations.

3. If Membrane's **SimpleRTSPServer** code can be adapted (it currently takes a file path and streams it [28]), we might use that as a template: instead of reading from file, connect it to our live source. Possibly, we could even "trick" it by feeding a named pipe or something, but better to modify it properly. Given that Membrane Simple RTSP Server v0.1.4 exists (Jan 2025) and v0.1.5 (Sep 2025) with presumably improvements, it's worth checking if it supports custom source injection. If not, we contribute or fork it to add that capability.

4. This is the most challenging part technically, but once solved, we have a fully native solution.

5. **Containerization:** Containerize the application. Use a multi-stage Docker build:

6. **Builder stage:** Based on an Elixir Alpine image (or install Elixir+Erlang on Alpine). Pull in Membrane deps (which will compile some native code like FFmpeg libraries for h264 parser, etc.). Also, build or install `libcamera` and `libcamera-apps`. Possibly we have a stage that uses Raspbian bullseye arm64 base to grab pre-built `libcamera-vid` binary if compiling is too slow; but if possible, compile in Alpine for consistency.
7. **Run stage:** Use Alpine minimal base. Copy the release from builder. Also copy any needed binaries (libcamera-vid, etc.) and shared libraries. We'll minimize size by only including what's necessary (the release will be mostly static, but libcamera likely has shared libs).
8. Add an entrypoint to run the Elixir release, and ensure it runs as PID 1 in the container.

9. We'll test the container on a Raspberry Pi by mounting /dev files and running it with `docker run` to ensure it finds the camera and streams.

10. **K3s Deployment:** Create a Kubernetes Deployment (or DaemonSet if one per host). Use node selectors to ensure it lands on the Raspberry Pi node. Include the needed volume or device mounts:

11. We may use the `devices:` field to pass in video devices. If k3s doesn't support that directly, using `privileged: true` in the security context is a fallback.
12. Set environment variables in the manifest for configurable parameters.
13. Optionally, setup a Service of type NodePort or ClusterIP + Ingress if we want ground station to reach it. In many UAV setups, ground and companion are on the same network (no need for K8s service abstraction, the ground station can directly hit the drone's IP and port). But for completeness, we document how to find the stream: e.g., "rtsp://<drone-ip>:8554/video" is the URL (with credentials if we set any).

14. We will also include a liveness probe: could be as simple as an HTTP GET on a small Phoenix server we run in the same app (since adding Phoenix might be heavy, an alternative is a custom TCP socket probe or just rely on the process running). Or we use `exec` probe to call a script that checks if the Membrane pipeline GenServer is alive. Simpler: we trust if the app hasn't crashed, it's likely fine – so a dummy TCP server could be coded to always respond "OK" if the app is up. This is a minor detail but useful for production.

15. **Testing & Tuning:** Deploy on the UAV or a representative environment and test with QGroundControl:

16. Measure latency: We can have the camera see a laptop screen running a timer and compare. We will adjust any GStreamer settings on QGC (like disabling its syncing or setting low latency mode) to truly get low latency.
17. Try various resolutions and frame rates via config to see how it affects latency and quality. We expect 720p60 might reduce motion blur for fast movement, aiding pilot control.
18. Monitor CPU usage on the Pi (should be low; the Elixir app itself might use <10% CPU when just piping data, most work on GPU).
19. Monitor memory – the BEAM VM plus buffers should be within reason (likely tens of MBs, which is fine on a 2GB CM4/CM5).

20. Check if any frames drop when two clients view simultaneously. The hardware encoder is doing only one stream, so it's fine; sending to two clients doubles outgoing bandwidth (~2x4Mbps=8Mbps which is okay on WiFi or ethernet). The Pi networking can handle that, but if using a telemetry radio, we might limit to one client in practice. We will note that multiple clients are supported, but using them may require sufficient link capacity.

21. We'll also test abnormal cases: restart the container while QGC is open (it should recover if stream comes back), disconnect and reconnect clients repeatedly (no port leakage or crashes), etc.

22. **Documentation:** Prepare documentation for users (pilots, ground station operators, and developers):

23. How to configure the stream parameters (env vars).
24. The RTSP URL format to use in QGC/ATAK (including any credentials if configured).
25. Troubleshooting tips (if video not showing, check firewall, or ensure QGC is set to RTSP and correct port; use VLC to isolate if issue is on drone or GCS side).
26. Note about enabling camera and setting GPU mem on the Pi host OS (as gleaned from RPi docs: e.g. enable camera and set `gpu_mem` in config.txt) [9] .
27. Explain maintainers how to update Membrane or adjust pipeline if needed (for instance, if upgrading to a newer Membrane version, or adding the recording feature, etc.).

By following this plan, we will incrementally build a robust streaming solution that meets the requirements. The end result will be a containerized service that provides a **low-latency, hardware-accelerated RTSP video feed** from the UAV, with the flexibility and maintainability of a modern Elixir application.

Future Work and Considerations

Beyond the initial deployment, a few areas are earmarked for future improvement, as briefly touched upon earlier:

- **Onboard Recording:** As described, adding an MP4 recording of the video is a logical extension. This could be triggered by a ground command (e.g., start/stop recording) and implemented by branching the Membrane pipeline to a file sink [33] . Care will be taken to ensure file writes don't stall the pipeline (maybe write to a buffer or separate thread if needed). The recording could include telemetry data overlay in the future (like embedding timestamp or drone attitude as metadata), but that would require generating an overlay on the video frames, something Membrane can do with filters if needed.

- **Video Overlay or Computer Vision Hooks:** We might integrate an element that overlays information on the video (for instance, a timestamp or crosshair). Membrane could support a video filter where we inject OSD (on-screen display). However, doing this in real-time on the Pi might be CPU-intensive unless using the GPU. The Pi's hardware can do a rudimentary overlay via the legacy firmware; with libcamera it's less straightforward. This is a stretch goal and would be carefully evaluated for performance impact. Alternatively, an AI accelerator could analyze frames and we might want to send the results somewhere – we could tee the raw frames to an AI module (but that would require raw frames, meaning additional load). More likely, if needed, we'd run a separate pipeline in another container for CV using the same camera (one gets H264, another requests raw –

but the camera can't usually do both raw and encoded at full rate; so we'd then do analysis on either the H264 or on key frames grabbed via still captures as done in the multi-stream example [40]).

- **Adaptive Bitrate Streaming:** To truly handle variable network conditions, implementing adaptive bitrate (ABR) logic would be valuable. One approach is to generate multiple streams at different qualities and let the client (if it supports it) switch, but RTSP clients typically don't handle adaptive streaming automatically – that's more in the realm of MPEG-DASH or HLS which introduces latency. Instead, we could implement simple **network-based adaptation**: monitor packet loss via RTCP or even custom ping measurements, and if loss exceeds a threshold, automatically reduce the encoder bitrate or resolution, then perhaps raise it back when conditions improve. Membrane does have an ABR plugin (perhaps geared towards switching in HLS) [41] , but for RTSP a manual approach is fine. This would require dynamic pipeline changes as discussed. We will gather data in real flights to decide how urgent this feature is.

- **Switch to WebRTC or SRT for lower latency**: RTSP over UDP as we have is quite low-latency (our target is ~200ms). WebRTC could reduce that further (100ms or less) and provide better coping with network dynamics (error correction, etc.), but it's more complex to set up and not natively supported by QGroundControl (though ATAK might support WebRTC streams in plugins). SRT is another protocol focusing on reliable low-latency stream (supported by some GCS tools). Our architecture using Membrane keeps this door open – Membrane has WebRTC and SRT plugins [13] [42] , so we could potentially feed the same camera stream into a WebRTC server to a web client, for instance. This is outside our current scope but is a good future direction if we need to integrate with web dashboards or reduce latency further for a specific use case.

- **Scaling to Multiple Cameras:** If future UAVs have multiple cameras (say front and downward), we'd run multiple instances of the pipeline or one application managing two pipelines. Membrane can handle multiple pipelines in one app, and we can differentiate them by RTSP path (e.g., `/front` vs `/down`). The current design is easily extendable to that: just add another source element and corresponding endpoints. The main constraint will be hardware (two encodes at once – Pi can do two 1080p30 encodes possibly, but we'd confirm). The container might then need both cameras exposed (`/dev/video0` and `/dev/video1` etc.). This modular approach is far simpler than if we had, say, used `gst-launch` static pipelines, because adding a second pipeline wouldn't require shell juggling – it's just additional Elixir code.

- **Security Enhancements:** In future, we might secure the stream. The simplest would be requiring a username/password for RTSP (which RTSP supports in the spec). If using an external RTSP server, it has config for credentials (as seen in the v4l2rtspserver example, using `-U user:pass` [43]). In Membrane's native approach, we could implement Basic Digest authentication in the RTSP handshake if needed. Another angle is encryption – either use SRTP (secure RTP) or run RTSP inside a VPN/tunnel. Given our environment (likely a closed network), we postpone this, but it's good to note.

In conclusion, the chosen design using Membrane on Elixir and containerized on Alpine meets the immediate requirements for a low-latency, configurable HD video stream from the UAV. It leverages Raspberry Pi's hardware encoder to ensure performance and uses standard protocols (RTSP/RTP) for compatibility [1] . The design is **maintainable and extensible** – future features like recording or adaptive streaming can be added with minimal refactoring, thanks to Membrane's modular pipeline architecture. By following this PRD, the development team can build a solution that will provide drone pilots with a responsive video feed and developers with a codebase that is clear and robust. The end result will be akin to a custom "IP camera" running on our UAV, tailored to the needs of real-time drone operation.

Sources:

1. Software Mansion, *Membrane Framework Overview* – Highlights Membrane's support for protocols like RTSP and integration with Elixir apps [8].
2. Elixir Forum – *Membrane with Raspberry Pi Camera* – Notes that Membrane's Raspberry Pi camera integration uses GPU encoding (1080p30/60) on the Pi's VideoCore [5].
3. Membrane RpiCam Plugin Documentation – Confirms capturing video from the Pi camera via libcamera with H264 output [6].
4. WTIP Blog – *Raspberry Pi Camera RTSP with multiple feeds* – Shows an example using hardware H264 encoding and notes the 1920px limit for H264 encoding on Pi GPU [7], as well as enabling the camera and GPU memory split [9].
5. Membrane Core Plugins – Mentions available elements like RTP payloader, Tee, MP4 sink, etc., used in our design [14] [33].
6. Wowza Media Systems – *RTSP Protocol Explained* – Emphasizes RTSP's use in IP cameras and low-latency benefit [1] [3], aligning with our use case.

---

[1] [2] [3] [4] RTSP: The Real-Time Streaming Protocol Explained | Wowza
https://www.wowza.com/blog/rtsp-the-real-time-streaming-protocol-explained

[5] Is the Raspberry Pi Zero W powerful enough for Nerves + Membrane + video streaming - Nerves Questions / Help - Elixir Programming Language Forum
https://elixirforum.com/t/is-the-raspberry-pi-zero-w-powerful-enough-for-nerves-membrane-video-streaming/28401

[6] [37] GitHub - membraneframework/membrane_rpicam_plugin: Membrane rpicam plugin
https://github.com/membraneframework/membrane_rpicam_plugin

[7] [9] [25] [26] [34] [35] [36] [40] Raspberry Pi Camera RTSP Streaming with multiple resolution feeds - wtip.net
https://www.wtip.net/blog/2021/08/raspberry-pi-camera-rtsp-streaming-with-multiple-resolution-feeds/

[8] [30] Membrane framework
https://membrane.stream/

[10] libcamera - Alpine Linux packages
https://pkgs.alpinelinux.org/package/edge/community/x86/libcamera

[11] wjtje/libcamera-apps-alpine - GitHub
https://github.com/wjtje/libcamera-apps-alpine

[12] Boombox — a simple streaming library on top of Membrane
https://blog.swmansion.com/boombox-a-simple-streaming-library-on-top-of-membrane-307649c09d63

[13] [14] [19] [20] [22] [23] [24] [29] [33] [41] [42] membraneframework | Hex
https://hex.pm/orgs/membraneframework

[15] [16] [17] [18] Membrane.Rpicam.Source — Membrane rpicam plugin v0.1.5
https://hexdocs.pm/membrane_rpicam_plugin/Membrane.Rpicam.Source.html

[21] The core of Membrane Framework, multimedia processing ... - GitHub
https://github.com/membraneframework/membrane_core

[27] [28] GitHub - membraneframework-labs/membrane_simple_rtsp_server
https://github.com/membraneframework-labs/membrane_simple_rtsp_server

[31] [32] Membrane RTSP source - Membrane Forum - Elixir Programming Language Forum
https://elixirforum.com/t/membrane-rtsp-source/28461

[38] [39] membrane_simple_rtsp_server | Hex
https://hex.pm/packages/membrane_simple_rtsp_server

[43] Raspberry Pi h264 RTSP Low Latency Camera Instructions — Ben Software Forum
https://www.bensoftware.com/forum/discussion/3254/raspberry-pi-h264-rtsp-low-latency-camera-instructions