UPPSALA
UNIVERSITET
Group 3 Project Report

Tejas Chakravarthy, Andreas Hadjoullis, Sarafroz Bekmurodova, Hardik Sai Kotagiri

25th September 2025

Abstract

This report describes the design and implementation of a multimedia stream-ing service on a Raspberry Pi. The project combined open-source tools including ffmpeg, nginx, and Python-based web technologies to deliver ad-aptive video streaming. In addition to technical outcomes, the report reflects on the collaborative process, ethical considerations, and lessons learned.

Contents

Chapter 1

Introduction

In recent years, the proliferation of compact, low-cost computing platforms such as the Raspberry Pi has significantly broadened opportunities for experimentation in applied computer science. These platforms provide sufficient computational power to implement distributed systems, network services, and lightweight applications, while remaining accessible to students and researchers. This project investigates the design and implementation of a multimedia streaming service hosted on a Raspberry Pi, with the aim of understanding both the technical processes involved and the collaborative practices required to deliver a functioning system.

The project is situated within the broader context of online video consumption, where streaming has become the predominant medium. Large-scale providers such as Netflix and YouTube employ sophisticated architectures that ensure adaptive resolution, low latency, and high availability. By contrast, this project explores whether similar principles can be achieved on a constrained platform using only open-source software and modest resources. The exercise therefore serves not only as a technical prototype, but also as a means of learning about software integration, system configuration, and cross-platform accessibility.

Equally central to the project is the process of teamwork and knowledge sharing. Our group adopted a rotating role model in which members alternated as coordinator, developer, tester, and documentation writer. This arrangement ensured that responsibilities were distributed equitably, that all members were exposed to different aspects of the project, and that collaboration was reinforced through regular communication. Progress was tracked through scheduled milestones, allowing us to assess whether the system was developing in line with expectations.

The report documents both technical achievements and reflections on the collaborative process. Specifically, it describes the research and methodological choices underlying the implementation, the ethical considerations of deploying a streaming service, and the lessons learned during development. The structure of the report is as follows: Chapter 2 surveys related work and comparable implementations. Chapter 3 presents the methodological framework guiding the project. Chapter 4 considers ethical questions relevant to the design and deployment of streaming technologies. Chapter 5 outlines the details of the implementation, including architectural decisions and testing procedures. Chapter 6 summarises team contributions and role allocation. Finally, we conclude with Chapter 7.

Chapter 2

Background / Related Work

Streaming media technologies have undergone a series of evolutionary stages, beginning with file download-and-playback approaches and progressing towards adaptive streaming protocols that dominate contemporary usage. Initial attempts to improve interactivity and reduce latency, such as the Real Time Streaming Protocol (RTSP), provided a foundation for streaming but were hindered by compatibility issues and limited scalability. The subsequent emergence of HTTP-based streaming marked a turning point, allowing video segments to be delivered over standard web infrastructure.

Among the most influential protocols is HTTP Live Streaming (HLS), introduced by Apple, which divides video into small segments and provides adaptive bitrate switching. Its counterpart, MPEG-DASH, similarly enables adaptive streaming over HTTP, with comparable advantages in compatibility and scalability. Both approaches exploit existing Content Delivery Networks (CDNs), minimising deployment overhead.

Implementation tools have also matured. The open-source utility `ffmpeg` has become the standard for video transcoding, capable of converting raw video into multiple formats and resolutions. In parallel, lightweight web servers such as `nginx` have demonstrated reliability in serving large numbers of concurrent requests. When combined, these tools provide a practical framework for hosting adaptive streaming services on modest hardware.

Within the academic literature, research on multimedia streaming highlights trade-offs between quality, compression, and latency. Fielding's (1) work on architectural styles remains influential in framing discussions of scalability and modularity in distributed systems. Further studies emphasise the increasing demand for adaptive methods that respond to heterogeneous devices and variable network conditions.

At the level of educational practice, projects involving Raspberry Pi streaming are not uncommon. Prior initiatives have demonstrated the feasibility of using the device as a media centre or a server for small-scale deployments. However, relatively few projects integrate these technical exercises with structured reflections on teamwork, project management, and ethical responsibility. This report therefore positions itself at the intersection of technical exploration and collaborative learning.

Chapter 3

Methods

The methodological approach combined structured planning, iterative development, and reflective evaluation. At the outset, the group established a contract defining rotating roles, conflict resolution strategies, and communication channels. Progress checkpoints were agreed upon, including deadlines for research, environment setup, implementation, and testing. Meetings were held at regular intervals to monitor progress and adjust the plan where necessary.

3.1   Research and Design.

The group reviewed available tools for video transcoding, web serving, and user interface development. Alternatives such as GStreamer or VLC were considered, but ffmpeg was selected due to its maturity, documentation, and flexibility. Nginx was chosen as the primary web server, and Python-based web development tools were identified as the most appropriate means of producing a landing page for user access. This decision reflected both familiarity within the group and the suitability of Python for rapid prototyping.

3.2   Environment Setup.

The Raspberry Pi was prepared with a suitable operating system and network configuration. Package dependencies for ffmpeg, nginx, and Python libraries were installed. Accessibility testing was integrated into this stage to ensure that the service could be reached from both desktop and mobile devices.

3.3   Implementation Activities.

Tasks were distributed according to role. The developer focused on configuring ffmpeg and nginx, the coordinator worked on the Python-based landing page, the tester ensured cross-platform accessibility, and the documentation writer maintained the evolving report draft.

3.4   Testing and Evaluation.

Transcoding was validated through playback of different resolutions. The HLS playlist was inspected for correctness, and nginx logs were reviewed for error handling. Accessibility tests verified that media could be streamed through both local and wireless networks. Later tests introduced HTTPS certificates to ensure secure connections.

3.5   Reflection and Adjustment.

Throughout the process, the group evaluated outcomes in light of objectives. Hardware acceleration was attempted but limited by the Raspberry Pi's performance. Difficulties with certificate configuration delayed the introduction of HTTPS, but these obstacles ultimately reinforced understanding of security protocols.

Chapter 4

Ethical Considerations

Although the project did not involve human subjects, it engaged several ethical dimensions that required careful reflection.

First, streaming technologies inherently raise questions of copyright and content ownership. The group elected to use only openly licensed or self-produced media files to avoid potential infringement. This decision ensured that the prototype demonstrated functionality without contravening intellectual property law.

Second, privacy and security considerations shaped system design. Because the system operated as a network-accessible service, precautions were taken to restrict deployment to controlled environments. HTTPS encryption was configured to prevent interception of data in transit. These measures align with broader ethical imperatives to protect user privacy and prevent misuse of technology.

Third, the use of artificial intelligence tools in the preparation of the report was explicitly limited in accordance with institutional guidelines. Language models were used solely for spelling and grammar checking, with all substantive content generated and critically reviewed by the group members. This ensured both originality and academic integrity.

Finally, the collaborative dimension of the project required adherence to ethical standards of fairness and transparency. Rotating roles and a conflict resolution procedure were introduced to prevent unequal distribution of labour and to manage potential disagreements constructively. Although conflicts did not arise, the framework reinforced accountability and trust within the group.

Chapter 5

Implementation

5.1   Architecture overview.

The system is a small LAN media streaming service running on a Raspberry Pi. The three main components are: (1) an HTTP server (nginx) that serves static HLS content (.m3u8 playlists and .ts segments), (2) a lightweight Python backend (Flask) that accepts uploads, stores originals, schedules transcoding using ffmpeg and exposes a small API, and (3) a client-side frontend (single-page UI + a standalone player) that uploads media, lists available items and plays HLS streams. This separation keeps playback fast and simple (nginx serves static files) while Flask handles control, metadata and the CPU-intensive transcoding jobs.

5.2   Server and filesystem layout.

All web content and HLS output are placed under a single web root: /var/www/pi_streamer/media. Within that directory we keep original/ (uploaded source files), hls/<id>/ (transcoded HLS outputs for each media item), and static frontend files (index.html, player.html, styles.css, script.js). Nginx is configured to serve /hls/ from the hls subfolder and to proxy small API calls (/api/*) to the Flask process on 127.0.0.1:8080. File and directory permissions are set so the nginx user can read and write to the directories directly.

5.3   Media processing (FFmpeg -> HLS).

When a user uploads a file the backend saves the original to original/ and schedules a background transcode job. Our chosen workflow transcodes on

upload (pre-generate HLS) because it provides predictable playback and avoids CPU spikes at first play. FFmpeg is used to produce HLS VOD output: a playlist (playlist.m3u8) and fixed-length segment files (segment_000.ts, segment_001.ts, . . .). The basic command preserves the input resolution if no scaling filter is provided; to create variant renditions we run FFmpeg with -vf scale=-2:<height> and bitrate settings for each target (for example 720p @ 2000 kb/s and 480p @ 800 kb/s), then produce a master playlist that references per-resolution playlists. Typical FFmpeg options used are -c:v libx264 -preset veryfast -crf 23 and -c:a aac -b:a 128k with -hls_time 6 and -hls_list_size 0 for VOD.

5.4   Backend design and job orchestration.

The backend is a single-file Flask app with a tiny SQLite metadata store (media.db). Endpoints include POST /api/upload (accept file upload), GET /api/media (list media), GET /api/status/<id> (status), and /watch/<id> (redirect to the HLS playlist). Uploads are validated (secure_filename, extension checks) and saved with a UUID to avoid collisions. A ThreadPoolExecutor (configured with a low max_workers to protect the Pi's CPU, e.g., 1) runs FFmpeg tasks asynchronously so the upload response can return immediately with a status URL. The DB stores status transitions: queued -> processing -> ready | failed and the frontend polls the API to show progress.

5.5   Nginx configuration highlights.

Nginx is configured to: (1) serve static HLS content and player assets from /var/www/pi_streamer/media, (2) set correct MIME types for .m3u8 and .ts, (3) add permissive CORS headers during development (Access-Control-Allow-Origin: *) so mobile/desktop browsers can fetch segments, and (4) increase client_max_body_size and adjust timeouts to allow large uploads (e.g., client_max_body_size 2048M). The server block also proxies /api/ and /watch/ to the Flask backend and returns standard access/error logs for debugging.

5.6   Frontend (index + player).

The frontend is intentionally minimal and dependency-light. index.html (plus styles.css and index.js) provides an upload form with an XHR-based uploader that shows a progress bar, a media library rendered from /api/media,

and an "Open in player" link which redirects to player.html. player.html is a small standalone page that reads ?id=<id> from the URL, constructs /hls/<id>/playlist.m3u8, and performs the following procedure: if the browser supports native HLS (Safari) set video.src to the .m3u8; otherwise, use hls.js to feed segments via MSE.

5.7 Testing, troubleshooting and developer conveniences.

During development we disable aggressive caching for CSS/JS in nginx so edits are visible immediately. Common issues and fixes are covered in the repo notes-e.g., 413 Request Entity Too Large is resolved by increasing client_max_body_size and using curl -I to verify Content-Type headers (make sure CSS is served with text/css and .ico with image/x-icon). Logs /var/log/nginx/pi_streamer.* and FFmpeg stderr are used to debug transcode failures.

5.8 Deployment decisions, limitations and future work.

For the one-month project scope we run nginx + ffmpeg + Flask on the Pi host. This is robust for small LAN demos with a few simultaneous viewers. Limitations: CPU and storage constraints on the Pi make multi-bitrate live transcoding expensive. For heavier workloads we recommend offloading transcoding to a more powerful machine. Future improvements that would make the project stronger: automatic thumbnail generation, thumbnail and metadata UI and optionally switching to an RTMP/WebRTC ingest for low-latency live streaming.

5.9 Security.

HTTPS certificates were installed to provide encrypted connections, enhancing trust and preventing potential interception. Although configuration proved challenging, the exercise deepened the group's understanding of network security practices.

5.10   Testing and Performance.

The system was tested on both desktop and mobile clients, including access via local hotspots. Playback was evaluated for latency, resolution switching, and stability. The system functioned reliably for demonstration purposes.

5.11   General Challenges.

One of the main challenges in this project was the effort required to properly integrate several different software components into a single working system. While tools like Nginx, FFmpeg, Flask, and hls.js each serve a clear purpose, bringing them together highlighted many small incompatibilities and configuration details that were not immediately obvious from the start. Documentation was often fragmented, meaning we had to spend significant time searching through examples, community forums, and official manuals to understand how each tool should be configured in the context of our project. Setting up Nginx was particularly challenging, as its directory structure and default configuration varied between Linux distributions. On our system, our custom configuration would not take effect until we explicitly removed the pre-installed "default" server block, which otherwise took priority and returned 404 errors. These integration issues ultimately improved our understanding of how the underlying components fit together.

5.12   Summary.

The implemented system is a practical, end-to-end HLS streaming prototype: users can upload files via a simple frontend, the backend transcodes and produces HLS content with FFmpeg, nginx serves the static HLS artifacts to any device on the LAN, and the client UI supports standalone playback with cross-browser compatibility using the native HLS + hls.js fallback pattern. The design prioritizes simplicity, reproducibility and clarity while leaving clear paths for performance hardening and feature expansion.

Chapter 6

Contributions

Each member contributed to all aspects of the project, but specific responsibilities were distributed as follows:

Tejas Chakravarthy (Coordinator): Coordinated group activities, ensured adherence to milestones, and contributed to development of the Python-based landing page. Drafted sections of the report, particularly on methodology and introduction.

Andreas Hadjoullis (Developer): Implemented ffmpeg integration with nginx, configured HLS streaming, and attempted hardware acceleration. Documented technical aspects of the system architecture.

Hardik Sai (Tester): Conducted extensive testing across platforms, validated HTTPS configuration, and reported on playback performance. Assisted in final evaluation and refinement.

Sarafroz Bekmurodova (Documentation Writer): Drafted the report in LaTeX, synthesised related work, and ensured citation consistency. Supported interface evaluation and coordinated proofreading.

All members engaged in research, implementation, and preparation of the final presentation. Communication was maintained through online platforms, and a clear conflict resolution protocol was established, though not required. The balanced distribution of tasks reinforced fairness and provided all members with experience across different aspects of the project lifecycle.

Chapter 7

Conclusion

In this project we successfully implemented a media streaming service hosted on a Raspberry Pi, combining Nginx, FFmpeg, Flask, and hls.js into a working system. Users can upload media files, which are processed into HLS format and securely streamed over HTTPS to a variety of client devices. While we faced challenges with integration, configuration, and adapting to differences across environments, these difficulties provided valuable hands-on experience with networking, web servers, and media handling. Overall, the project not only met its technical goals but also deepened our practical understanding of building and deploying networked applications.

Bibliography

[1] Roy T. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine, 2000.