Raspberry Pi 5 RTSP YOLO Pipeline with Hailo8L

Project Plan for pi-live-detect-rstp: A Modular RTSP Video AI Pipeline for Raspberry Pi 5 (8GB) + Hailo8L Hat

---

Introduction

Building a high-performance, modular video AI pipeline on a Raspberry Pi 5 (8GB) with a Hailo8L AI Hat is a formidable but increasingly accessible task as edge AI accelerators and open-source software mature. This project, to be hosted at https://github.com/kyleengza/pi-live-detect-rstp, targets robust real-time video analysis by probing, collecting, and caching frames from an RTSP stream, then running optimized detection (TensorFlow & YOLO on Hailo8L), annotating detected objects, and re-streaming annotated frames via a new RTSP endpoint. To maximize flexibility and reliability, each stage is implemented as a modular service, orchestrated via an in-memory cache/database, with install/maintenance scripts and comprehensive documentation.
This report presents a detailed, component-wise project plan with practical implementation strategies, core technology options, best practices, and relevant updates as of September 2025, integrating insights from industry guides, technical blogs, community discussions, and authoritative documentation.

Table: Service Summary & Configuration

Service	**Default Configuration**	**Responsibilities**	**Cache Usage**	**Log Output**	**Configurable?**	**Rel Tec**
RTSP Probe	URL: rtsp://192.168.100.4:8554/stream	Health/availability checks, stream metadata	Reads/writes state	Text/daemon & stdout (debug)	Yes (URL, probe freq.)	Op FFr Me cus
Frame Collection	Resolution: 640x480, FPS: stream default	Frame grabbing, raw frame caching	Writes frames	Text/daemon & stdout (debug)	Yes (res., FPS, buffer size)	Op zep Pica
Detection	Model: YOLOv8-HEF, Backend: Hailo8L, Confidence: 0.5	Neural detection, result caching	Reads frames/ writes det.	Text/daemon & stdout (debug)	Yes (model, backend, params)	Hai Zoo low

Annotation	Format: bounding boxes, label overlay	Annotated frame product ion, write to cache	Reads det./fr ames, write	Text/daemon & stdout (debug)	Yes (overlay style, freq.)	Op cus
RTSP Re-stream	URL: rtsp:// localhost:8554 /annotated	Streams annotated frames over RTSP	Reads annotat ed frames	Text/daemon & stdout (debug)	Yes (URL, FPS, buffer)	Me FFr zep
In-Memory Database	Implementation : Redis (default), TTL cache, Python Dict fallback	Cache for all services	Central store	Text/daemon & stdout (debug)	Yes (backend, limits, keys)	Red ols, Pyt
Service Manager	N/A	Status reports, health checks, service ops	Reads cache states	Text/daemon & stdout (debug)	N/A	sys cus scr

Each component can be configured via its own YAML/INI/JSON file, fallbacks to sensible defaults, and is discoverable through the central README.
The following report expands deeply on the architectural, hardware, software, and operational topics that underpin this table, with analysis, references, and best practices for each phase of the pipeline.

RTSP Probe

Overview & Implementation

The RTSP probe is responsible for checking the liveness, metadata, and basic accessibility of the configured RTSP stream. This primitive task lays the groundwork for reliability by verifying connectivity, extracting metadata (resolution, codec, FPS), and logging issues for subsequent automated remediation or alerting[2].
Implementation Considerations:

- *URL Configuration*: Must be configurable via environment variable or config file, with a robust default (rtsp://192.168.100.4:8554/stream).
- *Health Checks*: Regular heartbeat checks at tunable intervals; probe results written to in-memory cache for downstream services to consult.
- *Metadata Extraction*: Use FFmpeg or OpenCV to parse stream headers for video resolution, codec, and framerate[3].
- *Error Handling*: Failures are immediately logged; the probe retries with exponential backoff and updates cache to reflect offline, pending, or online state.

Python Technology:

- OpenCV, Python FFmpeg bindings, or dedicated libraries such as python-rtsp-client or custom socket-based checks support in-depth probing[3].
- MediaMTX (rtsp-simple-server) is a recommended server for local development and relaying streams efficiently, widely used on Raspberry Pi and compatible with modern Pi OSes[5].

Security & Performance:

- Credentials (user/password) must be supported for streams that require authentication.
- Logs must sanitize credentials before writing to disk.
- Support for both UDP- and TCP-based transport; documentation and comments must warn about trade-offs (latency vs packet loss).

Challenges:

- Network instability on Pi networks.
- Supporting dynamic reconfiguration without service restart.

Integration with the Pipeline

A responsive probe is critical. Downstream services (Frame Collection, Detection) must query the cache to decide whether to proceed or await the stream to recover. The probe's cache entries can include structured metadata (resolution, codec, timestamp of last successful probe) for consistency checks.

Frame Collection

Overview & Key Choices

This service streams raw frames from the probed RTSP endpoint and stores them in the in-memory database/cache for later processing. It acts as a real-time buffer, decoupling frame grabbing from compute-heavy detection and annotation services.

Capture Strategies:

- OpenCV's cv2.VideoCapture is widely proven for direct RTSP frame capture and is highly compatible with Raspberry Pi OS (Bullseye/Bookworm)[3].
- For more advanced scenarios-especially where multiple resolutions or camera feeds are needed-Picamera2 or FFmpeg-driven collection with MediaMTX is modern and reliable[5].

Configuration:

- Default resolution: 640x480 (tunable in config).
- Frame buffer size: configurable to balance memory usage and processing latency.
- Frame format: NPArray (for OpenCV), optionally raw JPEG/PNG byte strings for interoperability.
- Capture FPS: default to native stream FPS, with option to downsample for performance.

Implementation Details:

- Each captured frame is timestamped and stored in cache with key pattern frame:<timestamp>.
- Support for circular buffering-evict old frames when the buffer limit is reached.
- Log statistics: current buffer size, dropped frame rate, and capture delays.

In-Memory Cache Usage:

- Frames are indexed by a unique cache key; additional metadata (timestamp, frame ID, hash) supports deduplication.
- The cache design must minimize latency; as such, local Redis or fast TTLCache (see cachetools) are recommended for Python[7].

Edge Considerations:

- OS-level performance may be gated by limited Pi RAM; use optimized array structures (e.g., Numpy) and avoid unnecessary data copies.
- Provide fallback mechanisms: if connection is broken, service attempts reconnection in a background thread, while logging failure reasons.

Detection: TensorFlow & YOLO (Optimized for Hailo8L)

Hardware Integration: Hailo8L

Hailo8L Overview:

- Hailo8L is a powerful neural processing unit (NPU) delivering up to 13 TOPS, connected via an M.2 PCIe HAT, and currently official for Raspberry Pi 5 only[9].
- Requires specific drivers and firmwares, installed via the Hailo AI Kit scripts and utilities (hailo-all, HailoRT, firmware packages)[9].
- For optimal reliability, active cooling is strongly recommended, and the official 27W USB-C Pi power supply is mandated to supply stable current[8].
- Model inference can be offloaded to the NPU using Hailo tools and Python/C++ APIs: for Python inference, pyHailoRT, YOLO-HEF, Ultralytics YOLOv8n in ONNX/HEF formats.

Model Conversion & Setup:

- The project should provide a Dockerfile or manual instructions for converting YOLOv8n.pt weights to ONNX and then to Hailo’s HEF format using the Hailo DataFlow Compiler (DFC) on an x86 Ubuntu machine (because DFC is not ARM-native)[11].
- After conversion, .hef models are transferred to the Pi and run via Hailo Model Zoo or the HailoRT API.
- Configuration must allow specifying the model path, detection threshold, and relevant pre-processing parameters.

Detection Pipeline:

- Service reads uncached frames from memory, runs inference on batches or sequentially, writes detection results (bounding boxes, class labels, confidence) into cache under keys det:<frame_id|timestamp>.
- FPS: Near 30 FPS is achievable for YOLOv8n models on Hailo8L with correct pipeline and lightweight annotation/streaming steps.
- The detection stage must support batch or per-frame detection, depending on the required latency and system throughput.

API & SDK Landscape:

- Currently, the Hailo8L Python SDK on Raspberry Pi 5 is in active development; C++ examples are widely available and should be wrapped as subprocesses or with a thin CTypes/Cython bridge until full Python support stabilizes[10].
- Ultralytics YOLOv8, Hailo Model Zoo, and HailoRT are synergistic for this project. Documentation and example flows are maintained by Hailo, Seeed Studio, and the community[9].

Optimizations:

- Quantization: Use quantized ONNX models (Hailo tools provide quantization support at export/compile phase).
- Pre/post-processing: Move as much preprocessing (resizing, normalization) and NMS post-processing as possible onto the Pi CPU/NPU thread, separate from annotation.

Failure Handling:

- Missed frames should be recorded.
- Detection errors are logged at both text file and terminal for maximum debuggability (in debug mode).
- Pipeline disables itself gracefully if hardware, driver, or cache failure is detected (logs error, maintains status dump).

Frame Annotation

Purpose and Core Functionality

Annotation transforms detection outputs into human-readable, visually informative overlays on the raw video frames. This service consumes frames and associated detection results from the cache, then outputs annotated frames ready for streaming.

Annotation Features:

- Bounding boxes (with color coding per class or confidence score).
- Text overlays (class label, predicted probability, per-object ID).
- Optional overlays: masks (for segmentation models), custom polygons, or region-of-interest highlighting.

- Configurable annotation style, supported via YAML or JSON config.

Reference Implementation:

- OpenCV is the de facto choice for efficient, flexible image and video annotation. It supports all required primitives: drawing rectangles, polygons, circles, and rendering Unicode text[13].
- For more complex overlays or font handling, the Pillow (PIL) library can be optionally integrated.

Pipeline:

- Waits for a new detection result and its associated frame (linked by timestamp or unique key) in cache.
- Draws overlays as per configuration.
- Stores annotated frame in cache with key ann:<frame_id|timestamp>.
- Optionally, can write annotated frame to disk cache for offline debugging (especially in debug mode).
- Supports rapid re-annotation if configuration is updated on the fly (hot reload).

Debugging & Visualization:

- Supports additional debug overlays (object ID, inference time, data pipeline timings).
- Annotator can be tested interactively as a standalone Python script, supporting both text file and terminal log output.

Re-stream (Annotated RTSP Server)

Design and Rationale

This component serves the annotated frames over a new RTSP endpoint (e.g., rtsp://localhost:8554/annotated), acting as the final stage for external consumption (viewers, storage, or downstream analytics).

Streaming Options:

- **MediaMTX (rtsp-simple-server):** The default and recommended RTSP streaming server for Pi, offering H.264/265 support, low-latency relay, and easy integration with Python scripts via FFmpeg or custom code[5].
- **Custom Python RTSP server:** For stand-alone use or for more tightly managed resource use, zephyr-rtsp or python-rtsp-server can be evaluated.[15]
- **FFmpeg Streaming:** Alternately, annotated frames can be piped to FFmpeg, which acts as an encoder and streamer to the MediaMTX server.

Configuration and Optimization:

- Stream URL configurable.

- Output resolution, FPS, encoder settings (H.264 bitrate, GOP size), and buffer size all controlled via config file.
- Support for authentication (username, password, IP whitelist).

Performance and Security:

- Real-time relaying requires pipeline synchronization-annotated frames must be “fresh” to minimize perceived latency.
- Security: Enforce strong passwords, offer the option to bind endpoint only to localhost, suggest reverse proxy/firewall for external exposure.[2]
- Document that for public deployment, SSL/TLS and authentication are required.

Failure Recovery:

- If upstream pipeline stalls, RTSP output can signal “no frame” via a blank frame or placeholder overlay.
- Error events logged for post-mortem debugging and continuous integration tests.

Caching and In-Memory Database

Cache Design: Central Role

A high-throughput, in-memory database is central to coordinating this multi-service pipeline. The cache holds probe state, raw frames, detections, annotated frames, and orchestrates inter-service communication in a modular, decoupled way.

Backend Options:

- **Redis (recommended):** Provides atomic operations, key expiry (TTL), pub/sub, and is proven at scale. The official redis-py library is well supported on Raspberry Pi 5.[17]
- **Python-only fallback:** For simple installs or in test mode, Python dictionaries or LRU caches (functools.lru_cache, cachetools.TTLCache) offer an in-process, lightweight solution but lose durability and some concurrency guarantees.[7]
- TTL per cache entry ensures stale frames are cleaned up and unexpected memory growth is prevented.

Cache Key Design:

- Probe State: probe:<url>
- Raw Frame: frame:<timestamp>
- Detection Result: det:<frame_id>
- Annotated Frame: ann:<frame_id>
- Service State: svc:<service_name>:status

Performance and Reliability:

- Redis should be run locally with minimal config; optional authentication is available.

- All cache operations must be guarded with detailed error handling and fallback to avoid pipeline hangs due to transient cache outages.
- System health script periodically audits cache for integrity and reports on buffer usage & memory health.

Logging Architecture

Best Practice Logging for Python Microservices

Requirements:

- All daemons/services must log to text files on the OS drive.
- Standalone scripts (run by user for testing or diagnosis) must output identical logs to both file and terminal.
- Debug mode is enabled by default and can be toggled via environment variable or config per service.

Logging Implementation:

- The Python logging module, with rotating file handlers and timestamp-based filenames, provides a solid base.[18]
- For daemons: All logs end up in /var/log/pi-live-detect/ with file per service, rotated daily and on overflow.
- For standalone scripts: Dual-handler config, set at startup-one StreamHandler to stdout, and one FileHandler to disk.
- Log format: [timestamp][service][level] message (optional metadata).
- Debug logs include per-frame timings, cache hit/miss rates, and frame/detection statistics.

Daemonization Considerations:

- Logging must be (re-)initialized after Python process daemonization so that new file handlers are attached to child processes; this is a common pitfall in daemon service logging.[19]

Debugging Experience:

- Debug logs enable tracing the entire pipeline for a single frame from probe to annotation to outgoing RTSP.
- In production mode, only INFO, WARNING, and ERROR level logs are written by default.

Install, Verify, Uninstall Scripts (Sudo-aware)

Install Script

Goals:

- Bundle all project dependencies, system packages, and Python environments in a single, human-readable script.
- Detect root/non-root invocation; if not run as root, re-executes itself with sudo (with helpful messaging).
- Installs and configures all project services with defaults, but allows advanced users to override using CLI flags or pre-defined config files.

Structure:

- Performs OS package installation (Python >=3.8, pip, FFmpeg, Redis, MediaMTX, Hailo8L drivers & firmware).
- Creates Python virtualenv(s) with all required pip packages (requirements.txt), using pip and requirements files as needed.
- Optionally, checks out and builds Hailo Model Zoo and DFC utilities if not present-or documents x86 dependency for compilation step[10].
- Registers systemd service units (see below).
- Copies example configs and scripts to /opt/pi-live-detect-rstp and /etc/pi-live-detect-rstp/.
- Logs all steps to both terminal and disk, with clear errors on failures.
- Installs and enables systemd services by default.

Best Practices:

- Always prefer running the *whole script* via sudo rather than interspersing sudo calls within, both for user experience and power user flexibility.[21]
- If OS-level dependencies are missing (e.g., user has not enabled PCIe for M.2 Hailo), script should check and provide actionable instructions.
- For interactive installs, offer to download Hailo firmware only after EULA acceptance.

Verify Install Script

- Runs a suite of diagnostic checks: system info, hardware enumerations, service health (via systemctl), cache reachability, and a basic pipeline test (probe → detect → annotate a dummy frame).
- Logs a detailed status report and troubleshooting guide if issues arise.
- Dumps current config, recent log tail, and systemd/service status into a single status dump file for user support/debugging.

Uninstall Script

- Clearly warns before purging all project files, logs, service configs, and cache data.
- Stops and disables all systemd services, removes service unit files, project files, virtualenvs, and optionally Hailo drivers or related packages.

- Employs “trap cleanup” patterns for rollback if uninstall fails midway.[23]

Service Management (systemd)

Systemd Integration

- Each pipeline stage is packaged as an independent systemd service (unit file), defined with clear dependencies, restart policies, working directories, and environment settings.
- Main services:
  - rtsp-probe.service
  - frame-collector.service
  - detector.service
  - annotator.service
  - rtsp-restream.service
  - cache.service (if using local Redis)
  - service-manager.service (status, health, orchestration)

Systemd Best Practices:

- Use Type=simple for most services; if needed, add readiness notification with Type=notify.
- Restart=on-failure is default; services should be robust to intermittent error conditions.
- WorkingDirectory is explicitly set to the project root or where logs/configs are written.
- All custom services are documented in the README for manual status checking, restart, and debugging.
- User-level and system-level service options are possible; system-level is preferred for reliability and access to OS resources.[25]

Service Health and Status Dump

- service-manager regularly polls service health via systemctl and cache checks, exposing a diagnostic endpoint or status file in /var/log/pi-live-detect/.
- On failure, logs to both text file and terminal (if possible), and attempts safe shutdown or restart cascade as needed.

GitHub Documentation & Versioning

README as a Manual

README.md will be a comprehensive, user-friendly manual, covering:

- Project overview and feature set
- Hardware prerequisites, assembly instructions, and OS best practices (with links to official Raspberry Pi and Hailo guides)[8]
- Step-by-step install, upgrade, and uninstall instructions
- Detailed configuration reference with sample YAML/JSON/INI configs
- Service explanation with pipeline diagram (ASCII/art, or link to images)
- Troubleshooting guide for common issues (broken probe, no RTSP output, Hailo errors, out-of-memory, etc.)
- Usage examples (e.g., viewing annotated stream in VLC)
- Guidance for extending/customizing the pipeline (adding new detectors, changing the cache backend, etc.)

Markdown Features:

- Employs GitHub-flavored Markdown for headings, code blocks, bold/italic for emphasis, and tables for configuration matrices.[28][30]

Changelog Best Practices

- **CHANGELOG.md** conforms to community and Common Changelog principles: each release under a semantic version (vX.Y.Z), grouped entries under **Added**, **Changed**, **Fixed**, and **Removed**, with dates and commit links.[32]
- Changelog is updated via PRs; contributions must include a concise change entry.
- Major breaking changes, hardware/driver updates, or Hailo software release milestones are noted with extra prominence.
- Changelog includes upgrade notes for delicate operations (e.g., model format migrations, hardware driver changes).

Security and Performance Considerations

RTSP Security & Reliability

- Default configuration binds RTSP output stream to localhost; exposing to other networks requires explicit configuration and binding.
- Recommends strong authentication, and, for Internet exposure, SSH or VPN tunneling rather than directly exposing Pi.
- All logs strip sensitive fields before writing.
- RTSP server config supports client access controls, session timeouts, and limits on simultaneous clients.[2]

Pipeline Performance

- Log backlog statistics (frame latency, dropped frames, cache hit rates).
- System health scripts alert if RAM or CPU utilization exceeds expected values, or if the frame collector falls behind real time.
- Annotator and restream services must be optimized to avoid becoming new bottlenecks-use hardware accelerated codecs when possible, and gracefully degrade if hardware support is not detected.

Advanced Extensibility

Modular Microservices Design

- Each service is a standalone Python module, managed via systemd, communicating exclusively via the in-memory cache/database. This isolates failures and enhances maintainability-even enabling containerization (e.g., Docker Compose) for advanced users.
- Inter-process communication via cache keys and serialized payloads (e.g., JSON), leading to language or runtime agnosticism for future extensions (C++ detectors, Go streamers, etc.)[34].
- Service discovery is centralized via config files and the README.

Python Dependency Management

- All dependencies pinned with version numbers in requirements.txt, auto-generated via scripts that collect sub-dependencies (pipdeptree, pip show)[36].
- Install script checks for virtualenv creation and dependency installation idempotently.
- README includes troubleshooting for common Python/package/OS-level issues.

Conclusion

The **pi-live-detect-rstp** project represents a state-of-the-art, modular AI-powered video analysis pipeline for edge deployments on Raspberry Pi 5 with Hailo8L acceleration. Each service-probe, collector, detector, annotator, restreamer, and cache manager-has been outlined with up-to-date, field-tested implementation strategies, robust logging/debugging, and best-in-class install/maintenance tooling. Security, performance, and extensibility are first-class concerns, and the planned documentation ensures that users ranging from hobbyists to professionals can deploy, operate, and extend the system with confidence.
For development, GitHub will house not only the source, but richly detailed markdown documentation (README), a human-friendly changelog, install/verify/uninstall scripts, and community support hooks. The architecture is future-proof, welcoming extensions for alternate NPU/TPU hats, new detection models, and integrations with IoT systems or cloud endpoints.

By adhering to these principles and leveraging a collaborative, open-source ethos, **pi-live-detect-rstp** is positioned as a blueprint for next-generation real-time video AI applications at the edge.

References (36)

1. *RTSP Streaming Protocol: The Complete Guide for Developers (2025)*. https://www.videosdk.live/developer-hub/rtmp/rtsp-streaming-protocol
2. *Read Frames from RTSP Stream in Python - Stack Overflow*. https://stackoverflow.com/questions/17961318/read-frames-from-rtsp-stream-in-python
3. *RealSense on Raspberry Pi 5 - GitHub*. https://github.com/dannyh147/Pi5-realsense
4. *Python Cache: How to Speed Up Your Code with Effective Caching*. https://dev.to/crawlbase/python-cache-how-to-speed-up-your-code-with-effective-caching-4c55
5. *Getting Started with RPI5-Hailo8L - General - Hailo Community*. https://community.hailo.ai/t/getting-started-with-rpi5-hailo8l/740
6. *hailo-rpi5-examples/doc/install-raspberry-pi5.md at main - GitHub*. https://github.com/hailo-ai/hailo-rpi5-examples/blob/main/doc/install-raspberry-pi5.md
7. *How to run custom model(Yolov8) in hailo8l(raspberrypi5+hailo8l)*. https://community.hailo.ai/t/how-to-run-custom-model-yolov8-in-hailo8l-raspberrypi5-hailo8l/1458
8. *GitHub - seapanda0/hailo-yolo-guide: Guide to Convert and Inference ...*. https://github.com/seapanda0/hailo-yolo-guide
9. *Select a bounding box on an image and annotate - Stack Overflow*. https://stackoverflow.com/questions/58753442/select-a-bounding-box-on-an-image-and-annotate
10. *zephyr-rtsp · PyPI*. https://pypi.org/project/zephyr-rtsp/
11. *Fastest way to keep data in memory with Redis in Python*. https://stackoverflow.com/questions/52298118/fastest-way-to-keep-data-in-memory-with-redis-in-python
12. *Logging HOWTO - Python 3.13.7 documentation*. https://docs.python.org/3/howto/logging.html
13. *Maintaining Logging and/or stdout/stderr in Python Daemon*. https://stackoverflow.com/questions/13180720/maintaining-logging-and-or-stdout-stderr-in-python-daemon
14. *How do I run a 'sudo' command inside a script? - Ask Ubuntu*. https://askubuntu.com/questions/425754/how-do-i-run-a-sudo-command-inside-a-script
15. *How to run a command before a Bash script exits?*. https://stackoverflow.com/questions/2129923/how-to-run-a-command-before-a-bash-script-exits

16. *running python script as a systemd service - Stack Overflow*. https://stackoverflow.com/questions/42735934/running-python-script-as-a-systemd-service
17. *Basic Syntax - Markdown Guide*. https://www.markdownguide.org/basic-syntax/
18. *Format Markdown Table*. https://tabletomarkdown.com/format-markdown-table/
19. *How to Keep a Changelog (+5 Examples) - Whatfix*. https://whatfix.com/blog/changelog/
20. *How To Build and Deploy Microservices With Python - Kinsta®*. https://kinsta.com/blog/python-microservices/
21. *How to find a Python package's dependencies - Stack Overflow*. https://stackoverflow.com/questions/29751572/how-to-find-a-python-packages-dependencies