RTSP Streaming and Object Detection on Raspberry Pi 5 with Hailo8L

Comprehensive Project Document: Headless Dual-Stream RTSP Object Detection and Tracking System on Raspberry Pi 5 + Hailo8L AI Hat

Introduction

In the rapidly evolving landscape of edge AI, leveraging a Raspberry Pi 5 with a Hailo8L AI Hat presents a powerful, cost-effective foundation for real-time, high-performance object detection and tracking solutions. This document details the end-to-end design and implementation considerations for a headless image detection and RTSP streaming system running on Raspberry Pi OS Lite (64-bit), managed remotely over SSH, and supporting two simultaneous RTSP streams. The requirements specify robust AI-driven object detection with unique per-object labeling (via class_uid), a RAM-based database for high-speed cache and probe status, annotated output, detailed debug logging, a network-accessible cache inspection portal, and a modern API/HTTP interface for configuration and operational control. Installers must ensure seamless setup and full cleanup for both installation and uninstallation, supporting an out-of-the-box deployment experience.
This comprehensive report explores the foundational hardware setup, recommended software stack and service architecture, database/caching options and strategies, the object detection pipeline (including tracking and labeling), service management, API and UI interfaces, installation and teardown automation, performance optimization, and security best practices. A final section discusses Copilot prompt engineering for automated full-system code generation, aligning with contemporary standards for developer productivity and rapid solution prototyping.

Hardware Setup

1. Raspberry Pi 5, 64-bit, Headless Configuration

The Raspberry Pi 5 (RPi5) offers considerable improvements over previous generations, featuring a quad-core Cortex-A76 CPU and up to 8GB RAM-more than sufficient for concurrent video and AI workloads. Operating headlessly-without a direct display, keyboard, or mouse-unlocks deployment flexibility and reduces the system footprint[2].
Key configuration details:

- **OS:** Raspberry Pi OS Lite (64-bit, Bookworm recommended), minimal image with SSH enabled and no desktop environment.
- **Remote management:** SSH is the primary access method.
- **Network:** Wired Ethernet is preferable for multimedia workloads; setup and manage static IPs for reliability.

Recent changes in OS security require explicit user creation at install; default pi/raspberry credentials no longer exist[3].

Initial setup steps:

1. Download the appropriate OS image (64-bit, Lite)[2].
2. Prepare SD card using Raspberry Pi Imager; pre-configure SSH (by adding an empty ssh file to /boot) and WiFi (with wpa_supplicant.conf if needed)[1].
3. On first boot, connect via SSH and update system packages:
4. sudo apt update && sudo apt full-upgrade
5. Enforce secure static IP configuration if reliability is critical.

2. Hailo8L AI Hat Integration

The Hailo8L, an edge-oriented AI coprocessor (NPU), provides up to 13 TOPS of acceleration for neural networks and is integrated via PCIe, using a compatible HAT+ M.2 adaptor[5]. Multiple HAT options exist (Pineberry Pi, Geekworm, etc.-each with physical/thermal nuances), but for headless, high-performance ops:

- Use the official Raspberry Pi AI Kit or AI HAT+ with the Hailo8L module.
- Connect the camera via the MIPI interface before installing the AI Hat to avoid cable issues[4].

Installation highlights:

- Enable PCIe Gen3 in /boot/firmware/config.txt for best throughput:
- dtparam=pciex1
  dtparam=pciex1_gen=3
- Use the official 27W USB-C power supply to ensure reliable operation under compute load[4].

3. Camera and Video Inputs

- Two RTSP sources required: can be two IP cameras, two USB webcams with encoder, or a mix (IP camera strongly recommended for stability and compatibility).
- For locally attached cameras (e.g., RPi Camera Module 3), can stream using rpicam-apps, with images piped to RTSP streamers[6].

Software Architecture

To meet the stringent requirements of concurrent, high-throughput streaming and inference, robust service compartmentalization and minimal OS overhead are crucial. Below is a modular microservice-oriented architecture, leveraging processes/services for isolation and scalability.

Table: Key Services and Responsibilities

Service Name	Responsibility	Technology
RTSP Stream Collector	Ingests video streams (RTSP/IPCam/USB/MIPI), decodes frames	GStreamer, FFmpeg
Object Detector	Runs inference on video frames using Hailo8L, outputs raw detections	HailoRT, YOLOv8s
Object Tracker	Assigns persistent class_uid to detected objects, tracks across frames	DeepSORT, custom UID
Cache Manager	RAM-based DB: stores probe status, frames, detection metadata	Redis/In-memory DB
Annotation Service	Overlays detection results and labels on video frames	OpenCV
RTSP Re-streamer	Streams annotated video via RTSP	FFmpeg, GStreamer
Debug Logger	Captures logs (system/service/debug), persists to OS log system	Logging module/syslog
Cache Inspection Web	Lightweight UI/API to inspect live cache data and frame samples	FastAPI + WebSockets
Control API Server	Remote config, service lifecycle control, log retrieval	FastAPI, Flask, etc.
Installer/Uninstaller	Systemd installers, resource/cleanup automation	Bash, Python scripts

Each component is developed, deployed, and managed as an OS-level service, registered under systemd for robust lifecycle management[8][9].

---

1. Video Ingestion & Stream Management

- **RTSP Collector:** Uses GStreamer pipelines or FFmpeg for minimal-latency frame acquisition. GStreamer is highly recommended for complex streaming layouts and hardware integration [11].
  - For local camera: rpicam-vid to pipe into GStreamer or FFmpeg.
  - For RTSP: rtspsrc GStreamer element, or FFmpeg's -i with RTSP URL.

- **Parallel Streams:** Set up two separate collectors/processes, or a multistream pipeline (e.g., GStreamer's tee), ensuring independent buffering and error recovery[12].

2. AI Inference with Hailo8L

- **HailoRT** is the official software stack (user space library, kernel driver)[13]:
  - Supports compiled HEF models; the YOLOv8 family is recommended for a balance of speed and accuracy (use int8 quantized versions)[15][12].
  - Python and C++ APIs available; Python preferred for rapid development, C++ for maximum performance[16].
  - To enable dual-stream inference, spawn separate processes or manage multi-pipeline execution via batch APIs and multi-threading[12].
- **Model Preparation:** Convert (if custom) from PyTorch to ONNX to HEF using the official model zoo tools. Prebuilt COCO models are available; custom models must be annotated and exported following best practices[13].
- **Sample code:** Modify hailo-rpi5-examples for RTSP as per community guidelines and official forums[17].

3. Object Tracking and Persistent IDs

- **Unique object ID assignment:** Integrate DeepSORT (or similar) for multi-class, multi-instance tracking. DeepSORT is compatible with YOLOv8 and maintains a mapping of detected objects to unique internal IDs across frames, handling occlusion and re-identification[19].
  - Each detection is assigned both its class and a class_uid (persistent instance ID).
  - DeepSORT pairs Kalman filtering for positional prediction with appearance embeddings for association accuracy.
- **Association with labels:** Store mapping of class_uid ↔ per-frame detection data within the cache database for fast lookup. Ensure these IDs are unique per stream and resettable via API.
- **Alternative methods:** For lighter-weight scenarios where DeepSORT is overkill, implement a centroid-based tracker with Euclidean distance matching (OpenCV's centroid tracking)[19].

4. RAM-Based Cache

A fast, in-memory cache is crucial for low-latency querying and high-throughput storage of:

- Probe status (heartbeat, errors),
- Raw frames (and optionally, decoded JPEGs/thumbnails),
- Detection metadata (bounding boxes, class, IDs, etc.),
- Annotated frames (visual overlays).

Solutions:

- **Redis** is highly recommended: native in-memory operation, easy query API (via TCP), supports RAM size limits and expiry policies, and is mature with rich Python bindings (redis-py)[20].
    - Bind to local/private network only for security.
    - Set maxmemory and maxmemory-policy as appropriate for resource management.
    - Optionally persist periodic snapshots to disk for resilience.
- **Alternatives:** In-process RAM cache via Python dict, or use SQLite in-memory, but Redis offers the best mix of networked access, scalability, and inspection features[22][23].

Integration: Services connect to Redis to store/retrieve frame buffers, status, and detection records. Annotated frames can be stored as binary blobs with a TTL.

5. Debug Logging to OS

- **Structured, persistent logging:** Use the OS syslog facility or logging frameworks (logging module in Python, journald integration for systemd services).
    - Log relevant events: service startup, shutdown, errors, frame drops, AI inference stats, tracker errors, API access logs, and system resource utilization.
    - Separate logs for each microservice; consolidate and rotate logs with logrotate if necessary[25][26].

Network-Accessible Cache Inspection

A dedicated service enables interactive inspection and debugging of the current in-memory cache, offering:

- Recently received frames (raw & annotated),
- Per-stream stats (frame rate, detection count, error rates),
- Live probe status and event logs.

Implementation approaches:

- **HTTP API (FastAPI):** Serve JSON summaries, key-value lookups, and optionally preview images via HTTP endpoints. FastAPI is performant and straightforward for Raspberry Pi deployment, with built-in interactive docs and easy WebSocket integration for live data streaming[28].
    - Expose endpoints such as /cache/probes, /cache/detections, /cache/frames/{id}.
    - Optionally, offer protected web UI with live-updating stats/charts (using JS front-end such as Vue or React).
- **Authentication/Security:** Restrict API access (see "Security Practices" below). Only allow access from authorized subnets or use HTTP basic auth or token mechanisms[30].

HTTP API/Remote Configuration & Service Control

A unified HTTP API (e.g., FastAPI or Flask) provides:

- Remote service configuration: update RTSP sources, detection/class ID filter thresholds, frame rates, logging verbosity, tracker options, etc.
- Service lifecycle operations: start/stop individual services, reload configuration, trigger full health checks or restarts.
- Log retrieval: query last N log lines from each service, or fetch filtered error logs.
- System stats: memory, CPU, temperature, Redis usage, etc.

Key practices:

- Use RESTful API design for stability and client compatibility.
- Document all endpoints (auto OpenAPI docs with FastAPI).
- Enable CORS only as necessary.
- Integrate basic authentication or API tokens for remote actions, especially if the API might be exposed outside a secured network segment[29].

Example endpoint taxonomy:

GET /config -- Get current system configuration
POST /config -- Update system configuration
POST /service/restart -- Restart a service
GET /log/{service} -- Download recent logs
GET /status -- System/health summary
GET /cache/* -- Inspect cache contents

Caching Strategy

- **Short-term storage only:** As a RAM-based cache, data is not persisted long-term; rolling buffer with TTL per item to avoid memory exhaustion (Redis makes this easy).
- **Key design:** Use composite keys (e.g., stream1:frame:12345, stream2:detection:6789) with automatic expiry upon reaching desired cache window (e.g., last 10 seconds or 300 frames).
- **Large objects:** Store annotated frames as compressed images (e.g., JPEG/PNG) to minimize RAM usage.
- **Metadata storage:** Detection/index metadata is stored as JSON for rapid lookups and to power the cache inspection API.
- **Probe status (heartbeat, errors):** Store as dedicated fields or hashes in the database, and expire them on disconnection.

Object Detection Pipeline

1. Stream Capture

Each RTSP stream is decoded to raw video frames. Frames are timestamped and pipelined directly into the object detector for minimal latency.

2. Detection

- Leverage YOLOv8s or YOLOv8m quantized models pre-compiled for Hailo8L. Community results indicate 128-350 FPS is attainable at 640x640 per stream, far exceeding most RTSP frame rates[14][12].
- For CPU offloading, batch process frames on the Hailo8L in parallel or via pipelined workers.

3. Tracking

- DeepSORT integrates with YOLOv8s to maintain per-object unique IDs (class_uid). Each detection is paired with its assigned UID and tracked across frames, even through temporary occlusion[19].
- Output detections as JSON records: {"frame_id": ..., "uid": ..., "class_id": ..., "confidence": ..., "bbox": [...]}.

4. Annotation

- Use OpenCV to draw bounding boxes and overlay UID/class on each processed frame for visual inspection and RTSP re-streaming.

5. RTSP Outbound Streaming

- Encode annotated frames and re-stream over RTSP using FFmpeg or GStreamer. This allows downstream consumers to subscribe to live, labeled video as easily as to any IP camera.

Installation and Uninstallation Procedures

For clean deployment and robust lifecycle management, provide:

1. Automated Installers

- Offer a single installer script (e.g., install.sh) that:
  - Installs all system dependencies (Python, GStreamer/FFmpeg, Redis, HailoRT, required Python modules).
  - Sets up all services under systemd with correct service files, dependencies, and environment variables[8].

  - Optionally profiles hardware/system for best performance and applies mitigations (e.g., sets PCIe to Gen3).
- Use configuration files located in /etc/my-rtsp-ai-system/ for persistent settings; provide defaults for out-of-the-box use.

2. Uninstaller

- A matching uninstall.sh:
  - Stops and disables all relevant systemd units.
  - Removes installed binaries, virtual environments, config files, and service unit files.
  - Cleans up residual cache/logs.
  - Resets altered system state as needed (restoring default config if applicable)[9].
- Provide clear output during install/uninstall, and echo a summary of actions taken.

Deployment: Out-of-the-Box

- Ship with all services disabled except for the installer. On first run, enable auto-provisioning and bring up the API for configuration.
- Default configuration should connect to test/dummy streams (or use local camera if detected).
- Expose health check API on launch to verify system integrity.
- Provide clear documentation and/or web UI for initial configuration over the network (even from mobile devices).

Performance Optimization

Key considerations and techniques:

- **PCIe Gen3 activation** is crucial for maximizing Hailo8L throughput (double the FPS of Gen2) [15][12].
- **Offload all AI inference to NPU**; keep CPU available for I/O and service orchestration.
- **Use lowest feasible video resolution/FPS** for detection task to reduce memory and bandwidth, unless high-res needed for detection accuracy-experiment for optimal tradeoff.
- **Redis maxmemory** should match available RAM after OS/resource reservation.
- **Service isolation:** Run detector/tracker/annotator as separate processes to leverage all CPU cores (or as green-threaded async services).
- **Frame dropping:** If pipeline lags behind, drop oldest frames to prevent backlog.

- **Log sampling:** For high-frequency events, drop or aggregate logs to avoid disk I/O overload.

Community benchmarks confirm exceptional efficiency: up to 350 FPS (YOLOv6n) and 128 FPS (YOLOv8s) achievable on RPi5 + Hailo8L[14][12]; more than enough for two simultaneous RTSP streams of 25-30 FPS each, leaving headroom for system overhead.

Security Best Practices

SSH Access

- Enforce use of SSH key authentication; disable password logins[31].
- Regularly regenerate and update SSH keys; accept only 4096-bit RSA or Ed25519 keys.
- Harden /etc/ssh/sshd_config per best practices (disable DSA/ECDSA, enforce strong MACs, restrict root logins).

API Access

- Bind API to local interface or secure LAN subnet by default.
- Require HTTP Basic Auth or API key/token for configuration endpoints[30].
- Restrict HTTP access by firewall (e.g., through ufw, iptables) except on trusted network segments.
- Log all configuration and cache-inspection requests for postmortem audit.

System Services

- Run AI, tracker, and cache services as dedicated non-root users.
- Use systemd for predictable service restart and failure recovery.
- Regularly update all system packages and specific service dependencies to mitigate vulnerabilities.

Network Exposure

- Do not expose RTSP, cache inspection, or control APIs to the Internet unless tunneled over VPN with authentication.

Copilot Prompt Engineering: Kick-Start Full-System Development

Harnessing GitHub Copilot or similar AI pair programmers can radically accelerate initial codebase generation and ongoing iterative development. To maximize Copilot's efficacy, employ best practices in prompt engineering:
Effective Copilot Prompt Example:

Write a modular Python-based system for Raspberry Pi 5 on 64-bit Raspberry Pi OS Lite, orchestrating two parallel RTSP stream ingestors, each connected to an object detection pipeline. Integrate Hailo8L NPU for inference with YOLOv8s (using official HailoRT Python APIs). Use DeepSORT to track objects and assign a persistent class_uid to each tracked instance. Cache detection results, annotated frames, probe status, and logs using Redis as a RAM-only database; enforce per-entry TTLs to maintain a rolling cache window. Implement a FastAPI HTTP server exposing endpoints for configuration, cache inspection, and log retrieval; secure endpoints with HTTP Basic Auth. Each system component should run as a systemd service with clean install/uninstall scripts. Include a lightweight web dashboard (JS+HTML, e.g., using a single-file served by FastAPI) to inspect status and cache contents. Provide robust debug logging to OS syslog and support remote SSH management. Generate all `systemd` unit files, install/uninstall shell scripts, and comprehensive `requirements.txt`. Output all code in logically separated files with inline comments. Assume out-of-the-box deployment on a fresh Pi and test streams by default.

Prompt Construction Nuances:

- Be **explicit and specific**: define every subsystem you want, the libraries or APIs to use (e.g., HailoRT, FastAPI, Redis).
- **Declare system boundaries**: clearly indicate which services must be systemd-managed, how they interact, and installation prerequisites.
- **Describe expected interfaces**: APIs, endpoints, expected security posture (e.g., HTTP basic auth).
- **Specify output granularity**: per-file code generation, inline documentation, and logical structure.
- **Give example configs** when possible.

General Copilot Prompt Best Practices:

- **Start broad, then go specific**: State the use-case, then detail modules, then describe interfaces and non-functional requirements[33][34].
- **Avoid ambiguity**: Ask Copilot for distinct modules, not just “write the whole thing.”
- **Reference similar open source projects** to provide context.
- **State code output format preferences** (single script vs. directory structure, docstrings, etc.).
- **Iteratively refine**: Tweak prompts based on Copilot's responses to optimize for depth and accuracy.

Conclusion

This document offers a deeply detailed blueprint for a robust, headless, dual-stream RTSP object detection and tracking system built upon cutting-edge Raspberry Pi 5 hardware with Hailo8L AI acceleration. Each architectural choice and technology selection is grounded in the latest research, active project documentation, and real-world user experience from the AI, maker, and

open-source communities. Leveraging high-performance RTSP ingestion, Hailo's industry-leading inference acceleration, modern RAM-based caching, advanced object tracking, and a tightly integrated remote interface enables unprecedented efficiency and reliability for edge AI applications.
The included Copilot prompt blueprint empowers developers to accelerate prototyping or automate end-to-end generation of much of the required codebase, maximizing developer efficiency and allowing rapid translation from architecture to reality.

By following and adapting this comprehensive project document, system builders can deliver out-of-the-box ready, production-grade edge AI solutions-deployable in security, industrial, home automation, and research environments-while maintaining clarity, extensibility, and operational simplicity.

References (34)

1. *Raspberry PI OS Lite: Headless Install, Setup and Configure*. https://peppe8o.com/install-raspberry-pi-os-lite-in-your-raspberry-pi/
2. *New install RPiOS Lite, can't login ssh with default password*. https://forums.raspberrypi.com/viewtopic.php?t=335261
3. *Getting Started with Headless Raspberry Pi - TheLinuxCode*. https://thelinuxcode.com/install-headless-raspberry-pi-lite-64-bit-sd-card-raspberry-pi-imager/
4. *GitHub - hailo-ai/hailo-rpi5-examples*. https://github.com/hailo-ai/hailo-rpi5-examples
5. *How to connect a Hailo-8 to the Raspberry Pi 5*. https://community.hailo.ai/t/how-to-connect-a-hailo-8-to-the-raspberry-pi-5/183
6. *Getting started with RTSP Stream on RPi 5 - Raspberry Pi Forums*. https://forums.raspberrypi.com/viewtopic.php?t=378996
7. *linux - How to remove systemd services - Super User*. https://superuser.com/questions/513159/how-to-remove-systemd-services
8. *Script executed by a systemd service not deleting the service file*. https://stackoverflow.com/questions/70093462/script-executed-by-a-systemd-service-not-deleting-the-service-file
9. *How do I make a mosaic for multiple rtsp streams?*. https://stackoverflow.com/questions/50808214/how-do-i-make-a-mosaic-for-multiple-rtsp-streams
10. *Benchmark of Multistream Inference on Raspberrypi 5 with Hailo8*. https://wiki.seeedstudio.com/benchmark_of_multistream_inference_on_raspberrypi5_with_hailo8/
11. *Custom Object Detection on Raspberry Pi AI Kit with Hailo8L*. https://my.cytron.io/tutorial/raspberry-pi-ai-kit-custom-object-detection-with-h
12. *Raspberry pi 5 with Hailo-8L Benchmark*. https://community.hailo.ai/t/raspberry-pi-5-with-hailo-8l-benchmark/746
13. *Hailo8L on Raspberry Pi 5 in C++ - GitHub*. https://github.com/bmharper/hailo-rpi5-yolov8

14. *GitHub - MrYodaylay/hailo-rpi5-examples-rtsp*. https://github.com/MrYodaylay/hailo-rpi5-examples-rtsp
15. *Simple object tracking with OpenCV - PyImageSearch*. https://pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/
16. *How to Install Redis on Raspberry Pi - luisllamas.es*. https://www.luisllamas.es/en/install-redis-on-raspberry-pi/
17. *Setting up an SQLite Database on a Raspberry Pi*. https://pimylifeup.com/raspberry-pi-sqlite/
18. *View content of H2 or HSQLDB in-memory database*. https://stackoverflow.com/questions/7309359/view-content-of-h2-or-hsqldb-in-memory-database
19. *Raspberry Pi - Data Logging : 5 Steps (with Pictures) - Instructables*. https://www.instructables.com/Raspberry-Pi-Data-Logging/
20. *Use your Raspberry Pi as a data logger - Opensource.com*. https://opensource.com/article/22/3/raspberry-pi-data-logger
21. *Benchmark on RPi5 & CM4 running yolov8s with Hailo 8L*. https://forums.raspberrypi.com/viewtopic.php?t=373867
22. *Prompt Engineering Tips with GitHub Copilot - GeeksforGeeks*. https://www.geeksforgeeks.org/git/prompt-engineering-tips-with-github-copilot/
23. *Run a Server Manually - FastAPI - tiangolo*. https://fastapi.tiangolo.com/deployment/manually/
24. *Python HTTP Server with Basic Authentication - Raspberry Pi Forums*. https://forums.raspberrypi.com/viewtopic.php?t=24389
25. *Implementing Authentication Methods For A Restful Api On Raspberry Pi*. https://peerdh.com/blogs/programming-insights/implementing-authentication-methods-for-a-restful-api-on-raspberry-pi-3
26. *Raspberry Pi - SSH Hardening : 5 Steps - Instructables*. https://www.instructables.com/Raspberry-Pi-SSH-Hardening/
27. *Introduction to prompt engineering with GitHub Copilot*. https://learn.microsoft.com/en-us/training/modules/introduction-prompt-engineering-with-github-copilot/