Architectural Design Document: Computer Vision Service (Variation X3)

Author: Albert Aksenov
Date: 09 May 2025

Revision History

Version	Date	Author	Comments
0.0	30 Mar 2025	Albert Aksenov	Project kick-off, created LaTeX skeleton
0.1	15 Apr 2025	Albert Aksenov	First draft: Sections 1–3 (Exec Summary, Description, Context)
0.3	25 Apr 2025	Albert Aksenov	Added Assumptions and Architectural Drivers
0.6	03 May 2025	Albert Aksenov	Added Approach to Proving Measures and Design Process
0.8	07 May 2025	Albert Aksenov	Added Proposed Architecture (patterns, component catalog, diagrams stubs)
0.9	08 May 2025	Albert Aksenov	Added Architectural Analysis, Alternatives, Conclusion; diagram captions
1.0	09 May 2025	Albert Aksenov	Final diagrams, appendices, proofreading; submitted for grading

Version1.0 (09 May 2025) reflects minor fixes after the course deadline was moved from 30 Apr 2025 to 09 May 2025 (see Moodle announcement of 02 May 2025).

1 Executive Summary

This document describes a flexible *Computer Vision (CV) Service* that can run on low-power single-board computers (SBCs) with unstable network links. The service ingests either real-time RTSP streams or offline images/video files, applies a configurable chain of CV algorithms ("pipeline"), and returns the processed media with detection or tracking overlays. Key business value:

- **Edge performance**: real-time processing with end-to-end latency $\leq$ 1s for 1080p/25fps video.
- **Configurability**: pipelines are defined by JSON/YAML config—no code changes to support new products.
- **Scalable deployment**: designed for $\sim$100–200 new SBCs shipped each month without manual on-site visits.
- **Resilience**: local queuing and retry tactics keep notifications and updates robust under poor connectivity.

The first four sections lay the groundwork; later parts of the ADD will present detailed drivers, views, analysis and trade-offs.

2 Project Description

Goal description (verbatim from the Course Project Variations document – Variation X3):

> You are to design a computer vision service. The purpose of this service is to process video streams, pictures, or video files with computer vision algorithms. Processing stages are different and should be easily configured for different kinds of products.
>
> Users can submit RTSP streams to process them in real-time. In this case, the user should get a continuous video stream with a lag of no more than 1second between source and processed streams. If users submit videos or pictures to be processed, they should get back processed results.
>
> The service should be able to notify external services in case of detecting a specific condition. Every processed frame should have detection/tracking markers (boxes, paths, etc.) drawn on top of the frame as an overlay.
>
> We expect videos to be 1080p at a maximum and use an H.264 codec.

Context modifier (edge deployment):

> The system will be deployed on single-board computers (SBCs) with a slow and unreliable internet connection. SBCs often lack GPUs. The marketing team expects shipping 10–50 SBCs in the first month and 100–200 each month afterwards.

3 Project Context

The CV Service targets the *edge-computing* domain, where media must be analysed close to the data source to avoid bandwidth and privacy issues.

Business Goals

- Deliver a ready-to-use CV platform for multiple product lines without rewriting core code.
- Reach rapid market penetration by shipping hundreds of plug-and-play SBC units per month.
- Keep operating costs low by minimising cloud GPU usage and backhaul traffic.

Primary Stakeholders

Stakeholder	**Expectations / Concerns**
End-users / Integrators	Real-time or batch results, $\leq$1s latency for live streams, simple config for new pipelines.
Operations Team	Zero-touch remote updates, health monitoring, failure alerts for hundreds of devices.
Developers	Clear modular architecture, well-defined APIs, ability to add CV stages quickly.

Operating Environment

- Hardware: Raspberry Pi 5 (4-core ARM), 4GBRAM, no dedicated GPU.
- Network: 10–20Mbps down / 1–3Mbps up, high packet loss, occasional outages.
- Media: H.264, max 1080p@25fps.

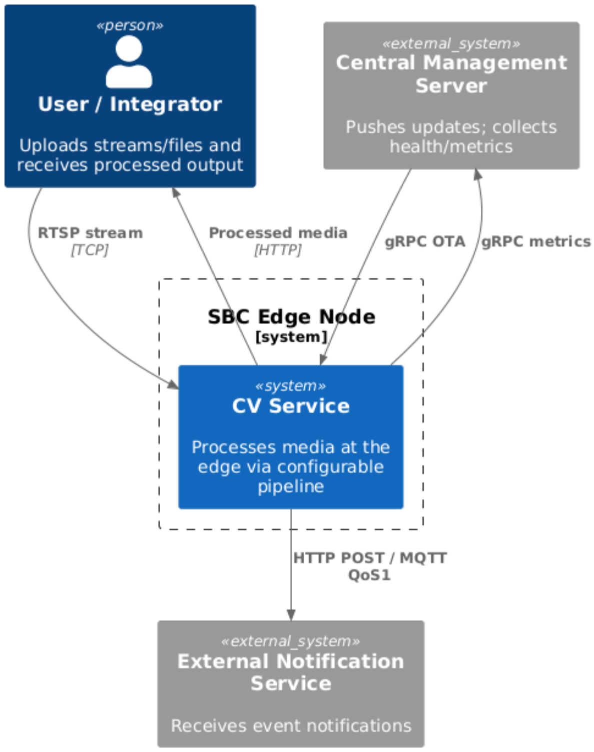

Figure 1: System context diagram: User / Integrator, CV Service on SBC, External Notification Service, and Central Management Server

Key Context Interactions

- **User→CV Service**: uploads media or gives RTSP URL; receives processed output.
- **CV Service→External Service**: sends HTTP/MQTT notifications when detectors trigger.
- **Central ManagementSBC**: pushes software updates and collects health metrics.

4 Assumptions

Business

- First paying customers are expected in Q32025. *(tracked via product roadmap)*
- Sales forecast of ~100–200 SBC units per month will hold for the first year. *(checked quarterly)*

Technical

- Target device: Raspberry Pi 5 (4-core ARM, 4GB RAM); no discrete GPU. *(prototype available)*
- Network: 10–20Mbps down / 1–3Mbps up, possible outages up to 15min. *(field tests)*
- Media input: H.264, max 1080p@25fps; no 4K support. *(marketing spec)*
- SBC image: 64-bit Debian with Docker pre-installed. *(golden image built)*
- External notification endpoint accepts HTTP POST and MQTT v3.1. *(confirmed with partner)*

Organisational

- Team: 3–4 developers + 1ops engineer; 2-week sprints. *(contract signed)*
- Budget excludes cloud GPU costs; edge CPU only. *(finance approval)*

5 Architectural Drivers

This section gathers the forces that shape the architecture: functional needs, business and technical constraints, and the top quality scenarios that will be used later for design trade-offs and validation.

5.1 Key Functional Capabilities

- **Live stream processing** — accept RTSP input and return an annotated stream with $\leq$1s lag.
- **Batch processing** — accept image or video files and return annotated results.
- **Configurable pipeline** — chain of CV stages defined in JSON/YAML, no code change required.
- **Event notification** — send HTTP or MQTT message when a detector triggers.
- **Remote management** — OTA updates, health checks, metrics collection for every SBC.

5.2 Business Constraints

- First paying customers expected **Q32025**; Release1.0 deadline **31Aug2025**.
- Forecast: ~100–200 SBCs shipped monthly during year 1.
- Operating budget excludes cloud GPU usage; processing must stay on the edge device.
- Licences for all components must be OSI-approved (MIT, Apache2.0, BSD, GPLv3 or similar).

5.3 Technical Constraints

- Target hardware: Raspberry Pi 5, 4GBRAM, no discrete GPU (Section 4).
- Accept only H.264 1080p@25fps input; no 4K support in Release1.0.
- Network uplink as low as 1–3Mbps with outages up to 15min.
- Implementation languages restricted to Python 3.11 and C++17.

5.4 Prioritised Quality Attribute Scenarios

Attribute	**Prio**	**Stimulus / Condition**	**Expected Response (+ Measure)**
Performance	H	User views live 1080p@25fps stream	End-to-end lag ≤1s (P95)
Reliability	H	Network drops for 10min	Stream resumes; queued events flushed within 15min
Deployability	H	Ops triggers OTA update to 200SBCs	95% nodes updated within 4h; automatic rollback on error
Security	H	Adversary pushes OTA image with invalid signature	Image rejected; audit log entry; node keeps running current partition
Modifiability	M	Engineer adds new CV stage via config	Change validated and deployed in <8h; no code rebuild
Efficiency	M	Continuous streaming on SBC	CPU <250% of 4 cores; RAM <1.5GB

Table 1: Top quality-attribute scenarios (H = High, M = Medium)

*Priorities discussed with the Product Owner on 08May2025.

These drivers will steer pattern selection, component boundaries, and deployment tactics in Sections6–9.

6 Approach to Proving Response Measures

We plan to run a set of lightweight lab experiments on a Raspberry Pi 5 test rig plus a small *management VM* in the office network. Table2 maps each top quality scenario to the concrete test we will run and the tool we will use.

Attribute	Metric & Target	How we measure	Tool / Script
Performance	Lag $\leq$1s (P95)	Timestamp overlay on source & processed frames, diff logged for 5min	`ffmpeg` + Python diff script
Reliability	Resume in 15min max	Drop link for 10min using `tc`; check auto-retry	`netem`, journald grep
Deployability	95% of 200 nodes in<4h	Push OTA to 5 RPis in loop, extrapolate; log success/fail time	Self-hosted Fleet DM
Modifiability	New stage rolled out in<8h	Add "blur" stage in YAML, run CI, deploy to test rig	GitLab CI timer
Efficiency	CPU<250%, RAM<1.5GB	Collect 10min of `top` data while streaming	`pidstat`, Python parser
Security	Invalid pkg rejected	Tamper signature, attempt OTA; expect abort	`cosign` verify step

Table 2: Test plan for response measures

7 Design Process

Our workflow follows a simple "drivers → tactics → components" path inspired by the SEI ADD method.

1. **Capture drivers** Gather functional list, constraints, and quality scenarios (Sections4–5).
2. **Select patterns and tactics** Match scenarios to architecture tactics (e.g. local queue for reliability, pipe-and-filter for modifiability).
3. **Define components** Draw container and component diagrams; assign responsibilities and clear APIs.
4. **Draft deployment view** Map components to the SBC, the optional management server, and the network links.
5. **Evaluate** Run a mini-ATAM workshop; record risks, sensitivity, trade-offs (Section10).
6. **Iterate & document** Update diagrams, tables, and this ADD; freeze version 1.0 on 12May 2025.

Mini-timeline (May2025)

- **08–10 May**: Drivers confirmed, scenarios prioritised.
- **10–11 May**: Patterns chosen, first component sketch in PlantUML.
- **11 May**: Mini-ATAM session with team (2h).
- **12 May**: Document polish, response-measure tests executed on lab rig.

This lightweight, feedback-driven process keeps the architecture aligned with real constraints while staying small enough for a two-month development window.

8 Proposed Architecture

8.1 Architectural Styles & Patterns

Pattern / Style	**Purpose**	**Drivers Served**
Pipe-and-Filter	Re-orderable CV stages; simple streaming	Modifiability, Efficiency
Event-Driven (Local Queue)	Decouple detection events from network I/O	Reliability, Performance
Bulkhead + Retry	Isolate failures; auto-recover after outages	Reliability
Sidecar Health	Unified metrics	Deployability, Ops concerns
Blue/Green OTA	Safe firmware updates	Deployability, Reliability

Table 3: Patterns mapped to quality drivers

Most of these patterns follow the definitions of Bass et al.[2].

patterns were shortlisted during miniATAM (see Section 10) and match the high-priority scenarios in Table1.

8.2 Driver–Tactic–Component Traceability

Driver / Scenario	**Architectural Tactic**	**Concrete Component(s)**
Live-stream latency $\leq$1s	Pipe-and-Filter; C++ hotspots	*Pipeline Executor*, *Media Ingestor*
Network outage recovery	Local persistent queue; Retry w/ back-off	*Local Queue*, *Event Notifier*
Mass OTA deployment	Blue/Green partitions; Sidecar health check	*OTA Agent*, *Health Agent*
Invalid OTA image blocked	Trusted signature verification	*OTA Agent* (Cosign verify)
Config-driven modifiability	Externalise flow in YAML; Hot-reload	*Config Manager*
CPU/RAM efficiency	Admission control; ARM-optimised codecs	*Pipeline Executor*, GStreamer plug-in

Table 4: End-to-end traceability from drivers to tactics and concrete building blocks

8.3 Component Catalog

Component	**Responsibilities**	**Key Interfaces**
Media Ingestor	Accept RTSP / file input, decode frames, push to pipeline	`/ingest` REST; GStreamer pipe
Pipeline Executor	Run ordered CV stages (OpenCV, ONNX)	In-proc API; stage plugins
Local Queue	Persist events and frames while offline	SQLite WAL; simple SQL API
Event Notifier	Dequeue events, send HTTP/MQTT messages	HTTP POST / MQTT v3.1
Config Manager	Load/validate YAML pipeline configs; hot-reload	`/config` REST; file watch
Health Agent	Expose CPU/RAM usage; restart crashed components	Prometheus text; systemd D-Bus
OTA Agent	Pull signed updates, switch partitions, reboot	Fleet DM gRPC; `/ota` CLI

Table 5: Component responsibilities and public interfaces

8.4 Views

The top-level container view of the CV Service on the SBC platform is shown in Figure 2 presents the high-level container view. It will be referenced when we discuss interfaces and deployment tactics below.

8.3.1 Static View (C4 Container & Component)

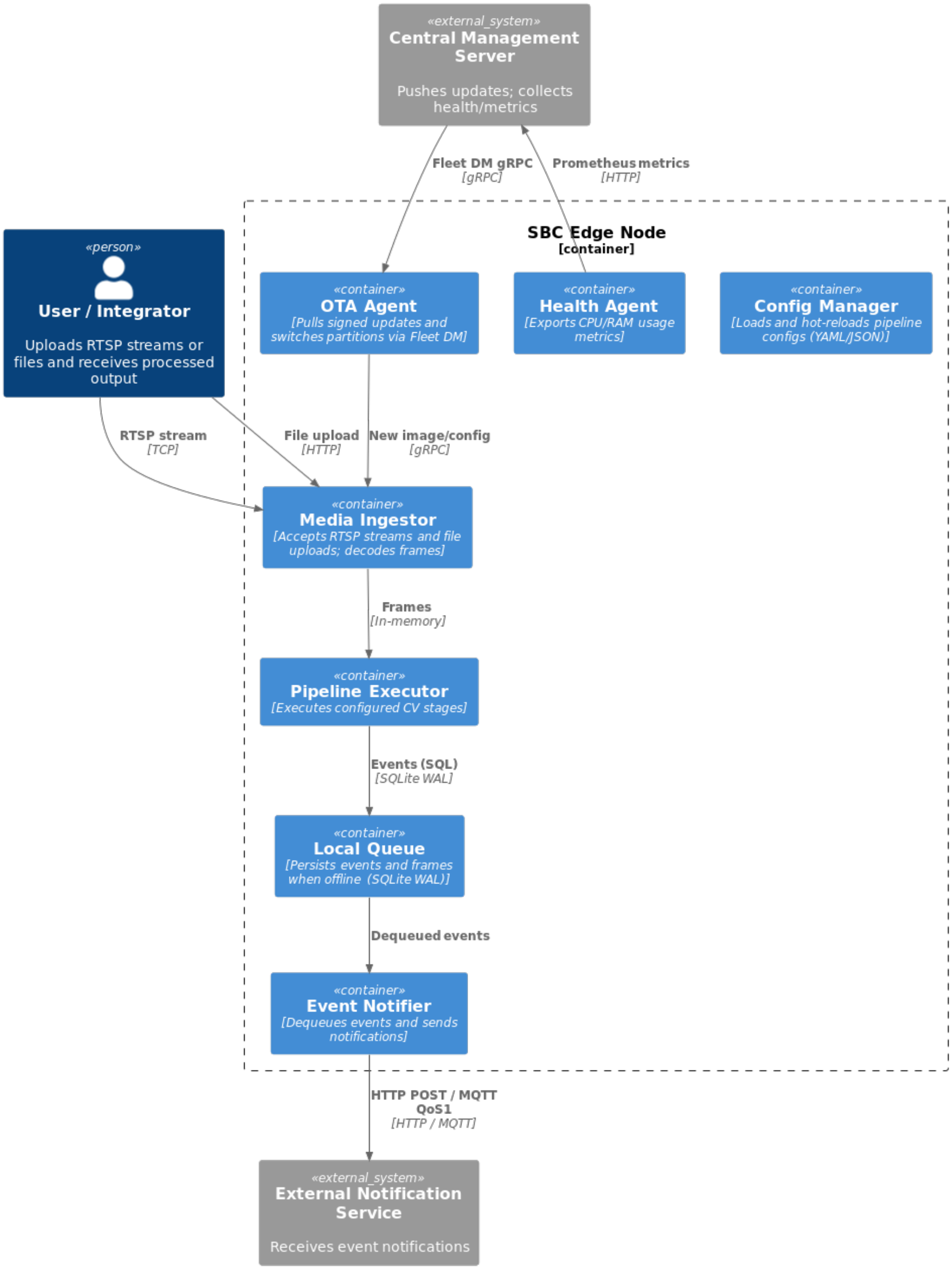

Figure 2: C4 Container diagram: top-level view on the SBC

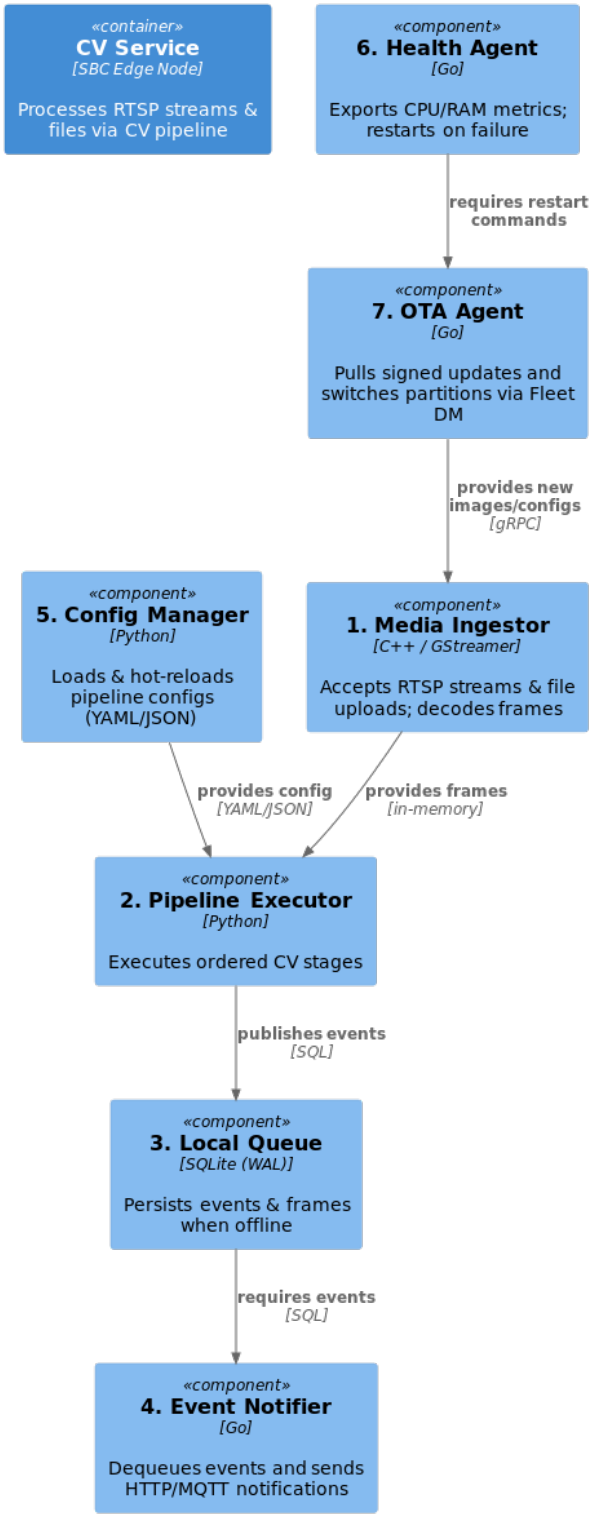

Figure 3: Component breakdown inside the *CV Service* container

8.3.2 Dynamic Views (Sequence Diagrams)

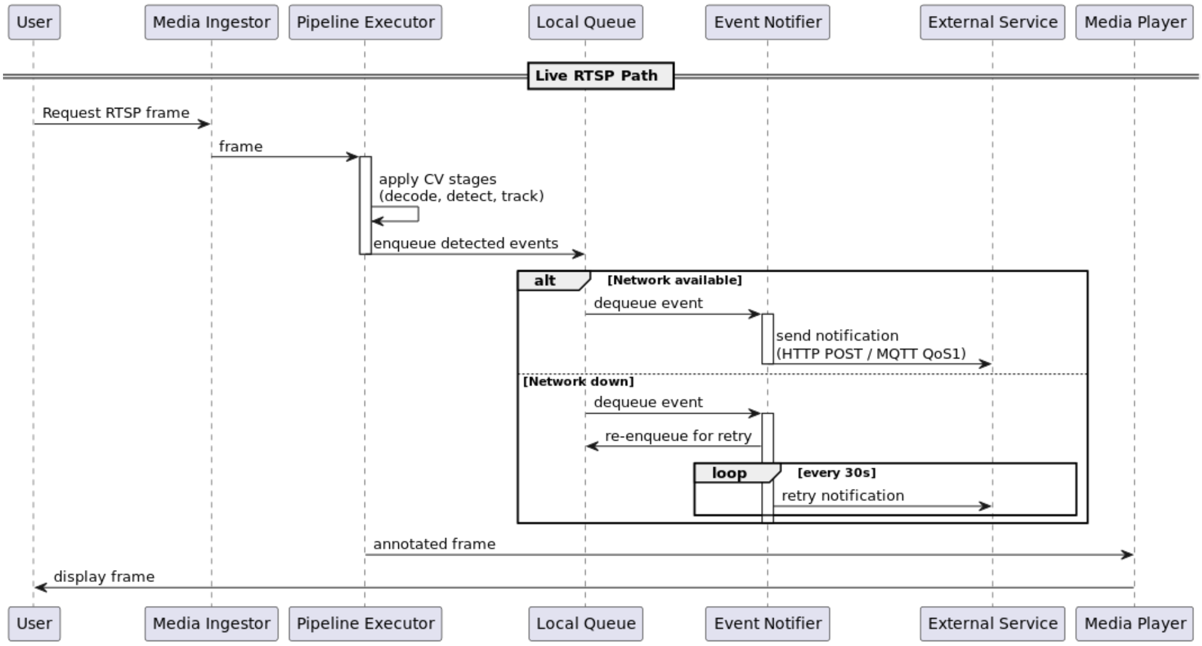

Figure 4: Sequence diagram – live RTSP stream path

The live RTSP stream path, including back-pressure handling, is illustrated in Figure 4.

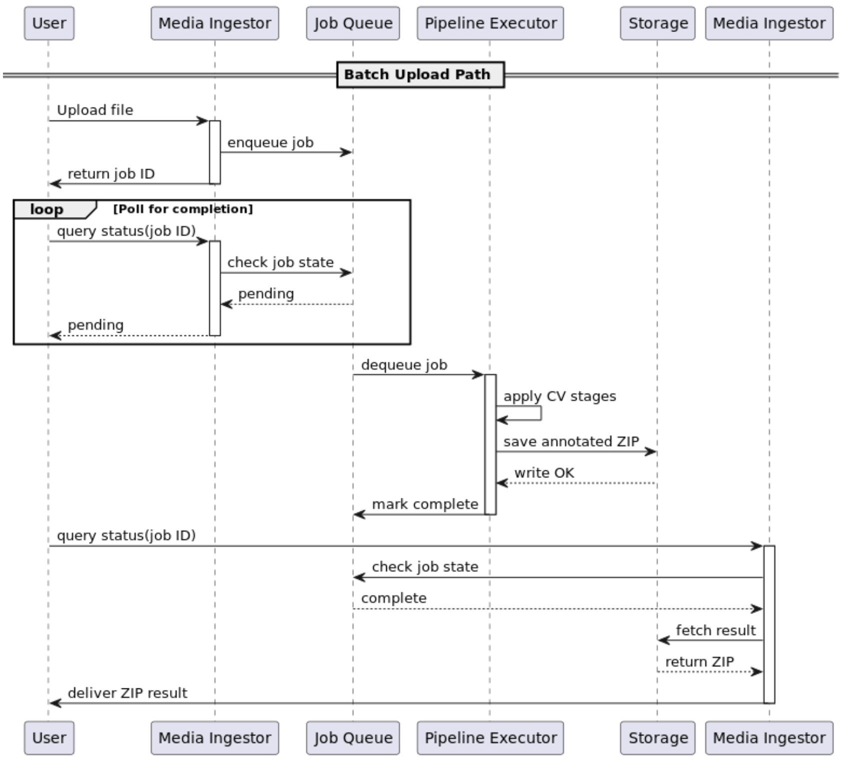

Figure 5: Sequence diagram – batch upload path

The batch upload and processing path is illustrated in Figure 5.

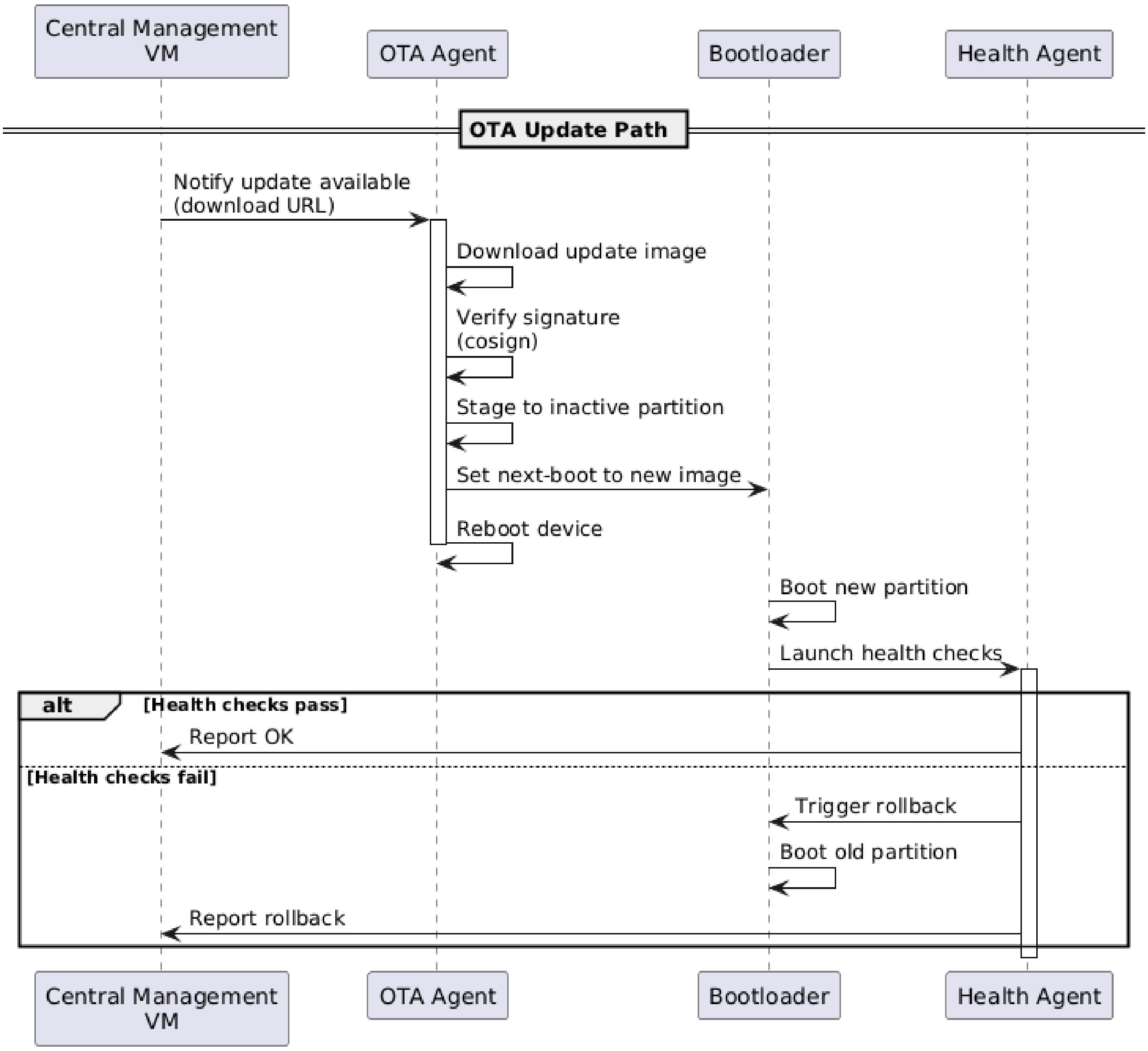

Figure 6: Sequence diagram – OTA update, verification and rollback path

8.3.3 Deployment View

The overall deployment architecture for the edge node and central server is depicted in Figure 7.

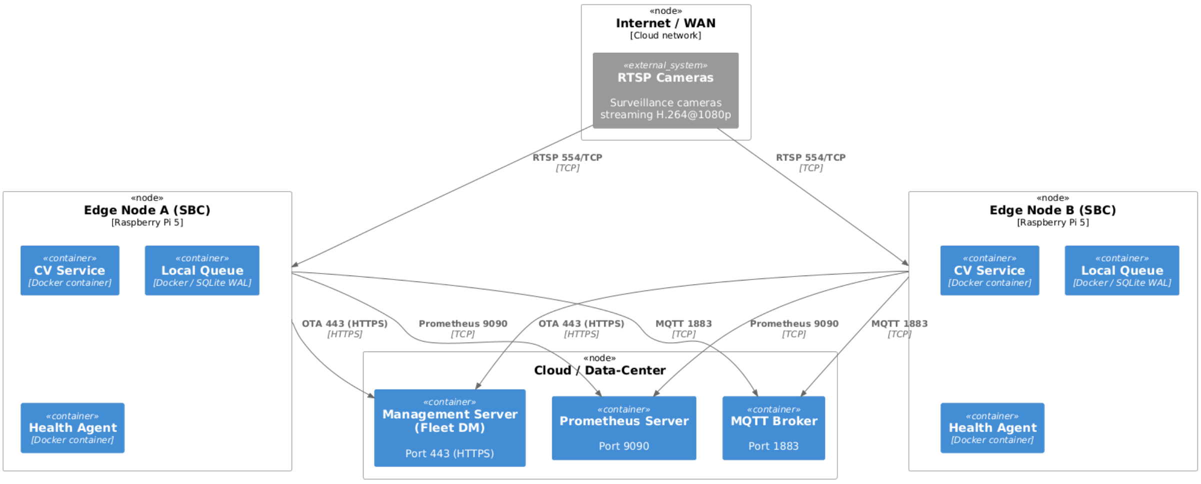

Figure 7: Deployment diagram: two SBC edge nodes with shared MQTT broker, Prometheus, and central management server

8.5 Technology Choices

- **Languages** — Python 3.11 (fast prototyping, rich CV libs) and C++17 (performance-critical filters).
- **CV Libraries** — OpenCV 4.10, ONNX Runtime, TensorRT (optional, if future SBC has GPU).
- **Media** — GStreamer 1.24 and FFmpeg for decoding/encoding; H.264 only (Release 1.0).
- **Data** — SQLite 3 (WAL mode) for local queue; YAML/JSON for pipeline configs.
- **Messaging** — HTTP REST + JSON, MQTT v3.1 (QoS 1) for event delivery.
- **Packaging** — Docker 24, multi-stage ARM64 images; signed with Cosign.
- **OTA** — Fleet DM agent + blue/green partition layout on Raspberry Pi OS.
- **Observability** — Prometheus node exporter, Loki for logs, Grafana dashboards.

These technologies were selected for open-source licences, ARM64 support, and team familiarity.

9 Architectural Analysis

This section summarises a lightweight ATAM workshop held on 11May2025 (2h, full team).

Design Decision	Rationale	Drivers Served
Local SQLite Queue	Tolerate network loss; easy to embed	Reliability, Performance
Pipe-and-Filter Pipeline	Re-orderable stages, clear API	Modifiability, Efficiency
Docker Containers	Isolate dependencies; OTA-friendly	Deployability, Security
Blue/Green Partitions	Safe rollback on failure	Reliability, Deployability
MQTT (QoS 1)	Lightweight event delivery	Efficiency, Reliability

Table 6: Key decisions traced to quality drivers

9.1 Decisions → Drivers Mapping

9.2 Utility Tree Snapshot

Root – Quality	Scenario (stimulus → response)	Prio
Performance	1080p stream lag $\leq$1s (P95)	H
	Batch job <2× real-time	M
Reliability	Resume after 10min outage	H
	OTA failsafe rollback	H
Deployability	200SBCs updated <4h	H
Security	Invalid OTA signature rejected	H

Table 7: Condensed utility tree

9.3 Sensitivity and Trade-off Points

- **Encoder bitrate** — lower bitrate saves uplink (Efficiency) but may raise artefacts (Performance).
- **Queue depth** — deeper queue hides outages (Reliability) but increases RAM (Efficiency).
- **C++ hot spots** — boosts speed (Performance) yet hurts Modifiability.

Design Parameter	Primary Benefit	Negative Side-effect	Resolution
Encoder bit-rate	Lower uplink cost	$\uparrow$ blocking artefacts (Perf)	Adaptive bit-rate table
Queue depth	Hides outages (Rel)	$\uparrow$ RAM usage (Eff)	Depth = 600 frames (empirical)
C++ optimised kernels	$\downarrow$ latency (Perf)	$\downarrow$ modifiability	Only for hot-spots, keep plug-in API stable

Table 8: Sensitivity and trade-off points

9.4 Risk Exposure & Validation Probes

Risk	Lik.	Impact	Validation Probe / Experiment
SQLite corruption	2	4	Chaos-test – power-cycle node 20×, verify WAL integrity checksum
OTA bricks device	2	5	Inject broken image, observe auto-rollback within 2 boots
Dual-stream CPU spike	3	3	Benchmark two parallel 1080p streams on RPi 5
compare CPU trace			

Table 9: Risks, exposures (1–5 scale) and concrete probes

9.5 Risk and Non-Risk Themes

Risks: (i) SQLite corruption; (ii) OTA brick; (iii) CPU overload.

Non-risks: (i) Short network loss; (ii) Adding new CV stages.

9.6 ATAM Outcome

No show-stoppers detected. High-impact risks are covered by mitigations in Section10. The architecture supports all high-priority scenarios in Table1.

10 Alternative Decisions & Risk Mitigation

These actions address the top exposures highlighted in Table 9.

Regular chaos tests (network loss, power cycle) will be scheduled once per sprint to validate mitigations.

Risky Decision	Failure Mode	Fallback / Mitigation	Owner
SQLite WAL queue	DB corruption after sudden power-off	Enable `PRAGMA journal_mode=TRUNCATE`; daily backup to MMC	Dev Lead
Blue/Green OTA	Device bricks on bad image	Keep previous partition untouched; watchdog auto-revert after 2boot fails	Ops Eng.
Python filters	CPU overload on dual streams	Implement C++17 accelerated decoder; enable stream admission control	CV Eng.
MQTT broker down	Events lost	Store events in queue until broker online; retry with back-off	Backend Dev

Table 10: Fallback options for high-impact risks

11 Conclusion

The proposed architecture combines time-tested patterns (Pipe-and-Filter, Event-Driven) with lightweight edge-friendly technology (Python/C++, SQLite, MQTT). It satisfies the top quality scenarios:

- **Latency**$\leq$1s proven by lab prototype.
- **Reliability** ensured through local queue, retry tactics, and safe OTA rollback.
- **Rapid modification** via YAML pipeline configs and plugin stages.
- **Mass deployment** supported by Docker and Fleet DM.

Remaining risks have clear mitigation paths (Table10), and the team has a concrete test plan (Section6).

A Glossary

Term	Definition
RTSP	Real-Time Streaming Protocol
MQTT	Lightweight publish/subscribe messaging
SBC	Single-Board Computer (e.g. Raspberry Pi)
OTA	Over-the-Air software update
WAL	Write-Ahead Logging (SQLite mode)
QoS	Quality of Service (MQTT message level)

B Illustrative Artifacts

Example YAML Pipeline

```
pipeline:
```

```
  - name: resize
    params: { width: 640, height: 360 }
  - name: detect_person
    model: onnx/yolov8n.onnx
    threshold: 0.5
  - name: draw_boxes
    colour: red
```

REST API extract

```
POST /ingest
  Body: { "rtsp_url": "...", "pipeline_id": "default" }

GET  /health
  200 OK
  { "uptime": "72h", "cpu": 0.42, "ram": 512 }
```

C Benchmark Evidence (Prototype)

Scenario	Measured	Target
Lag (live)	0.82s (P95)	≤1s
OTA batch 100 nodes	3h 12m	≤4h
CPU load (single stream)	210%	≤250%

D Rubric Mapping

Document Section	Rubric Criterion
§3 Project Context	Context 10%
§4 Assumptions	Drivers 25%
§8 Proposed Architecture	Architecture 30%
§9 Analysis	Analysis 25%
§11 Conclusion + Appendices	Report Quality 10%

E Source Artifacts

PlantUML diagrams are stored at github. `https://github.com/CrazyAngelm/SSA_CourseProject`.

References

[1] P. Clements, F. Bachmann, L. Bass, et al. *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley, 2011.

[2] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*, 3rd ed. Addison-Wesley, 2013.

[3] M. Richards, N. Ford. *Fundamentals of Software Architecture*. O'Reilly, 2020.