# ANSCORE — System Architecture (L2 + L3)

*Developer reference for the ANSVIS Communication Layer (CL) backend and the Qt/QML cross-platform desktop UI*

ANSCENTER

2026-05-26

## Executive summary

ANSCORE is a Vision-AI Video Management System split into three layers:

- **Layer 1 — Edge runtime** (per target). Owns cameras, AI inference, trackers, rules, triggers. Built into the existing `ANSCORE/modules/*` C++ tree (ANSCV, ANSODEngine, ANSFR, ANSLPR, ANSOCR, ANSMOT). For this document L1 is *abstract*: we describe only the interface it exposes upward to L2.
- **Layer 2 — ANSVIS CL** (one per customer). Aggregates targets, persists configuration, ingests events, brokers live media. Exposes a single REST contract (`/api/v1`) to L3. In v1 it is built as an **adapter layer in front of the existing ANSVIS server** — no rewrite required.
- **Layer 3 — Qt/QML desktop UI** (Win / macOS / Linux). The operator-facing application. Talks to L2 exclusively.

**Build assignment:**

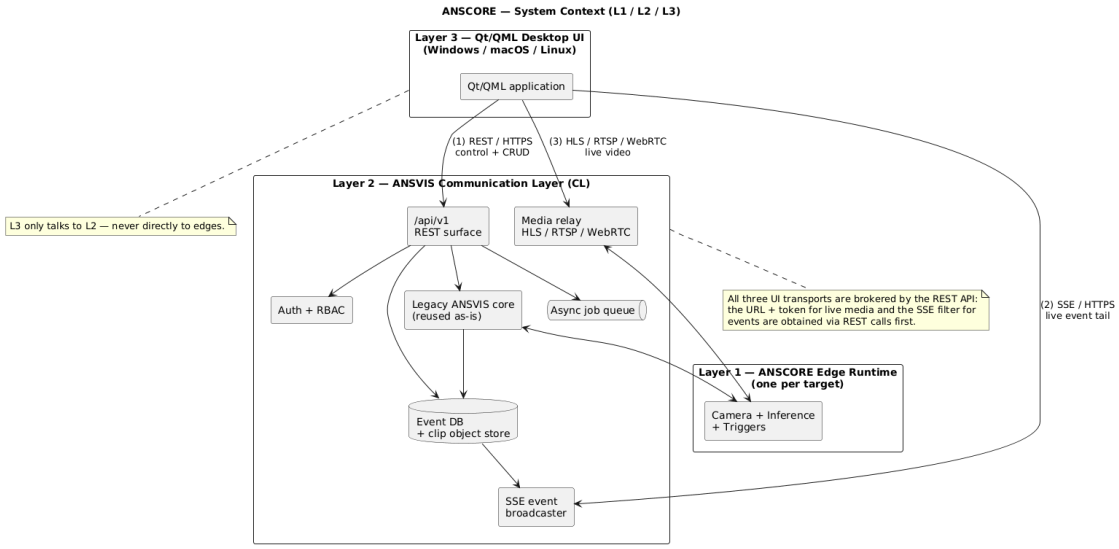
- L1 and L2 are built by the **core team** under architect-level review.
- L3 is built by a **6-engineer team**.

- L2 ? L3 is **REST + SSE + (HLS / RTSP / WebRTC)** — defined by `contracts/openapi.v1.json` (67 paths, 102 operations, 91 schemas).

The contract is the only artifact both sides block on. Once published, UI engineers develop against a Prism mock; the backend **core team** implements adapters per resource in parallel. Integration is the joining point.

# Part I — System overview

## The three layers




System context

L3 only ever talks to L2. L2 brokers everything else: configuration push down to edges, event upload from edges, media relay. This keeps the UI team fully decoupled from L1 velocity.

## Three transports between L3 and L2

| # | Transport | Purpose | Carries |
|---|---|---|---|
| 1 | REST over HTTPS | Control plane: all CRUD, configuration, deploy commands | JSON request/response, RFC 7807 errors |
| 2 | Server-Sent Events (SSE) over HTTPS | Live event tail (push of detections, target status) | `text/event-stream`, JSON per event |
| 3 | HLS / RTSP / WebRTC | Live video | Media segments; transport chosen per stream |

The REST API brokers the other two: the UI calls REST to obtain a stream descriptor or SSE endpoint URL.

## Why this split

The earlier proposal of using gRPC end-to-end was reversed because the existing ANSVIS server already exposes a REST API. Pragmatic v1: **reuse what exists**, adapt it into a standards-compliant `/api/v1` surface, ship faster. gRPC remains an option for a future v2 if needed.

## Build ownership at a glance

| Layer | Pace |
|---|---|
| L1 (edge runtime) | Slower — high complexity (frame bus, inference batching, plugin SDK) |
| L2 (CL) | Faster — modular per-resource adapters |
| L3 (Qt/QML UI) | Independent — mock-first development |

# Part II — Layer 1 (abstract overview)

L1 is the **edge runtime** that runs on each target. From L2's perspective it is a black box that:

- Receives **configuration** (cameras, AI models, AI tasks, rules, triggers) from L2 and persists it locally.
- Captures frames from cameras (RTSP / RTMP / SRT / ONVIF / simulated).
- Runs **AI pipelines** (object detection, FR, LPR, OCR, MOT, custom plugins).
- Evaluates **rules** and emits **events** with evidence (snapshot frame, optional clip).
- Dispatches **triggers** (HTTP, MQTT, email, push, broker integrations).
- Exposes **liveview** streams (RTSP / SRT / future WebRTC).
- Reports **health / metrics / logs** upward.

L1 ⍰ L2 protocol is out of scope for this document. For v1, L2's adapter calls into the legacy ANSVIS server's existing protocol with edges (this is what currently works in production).

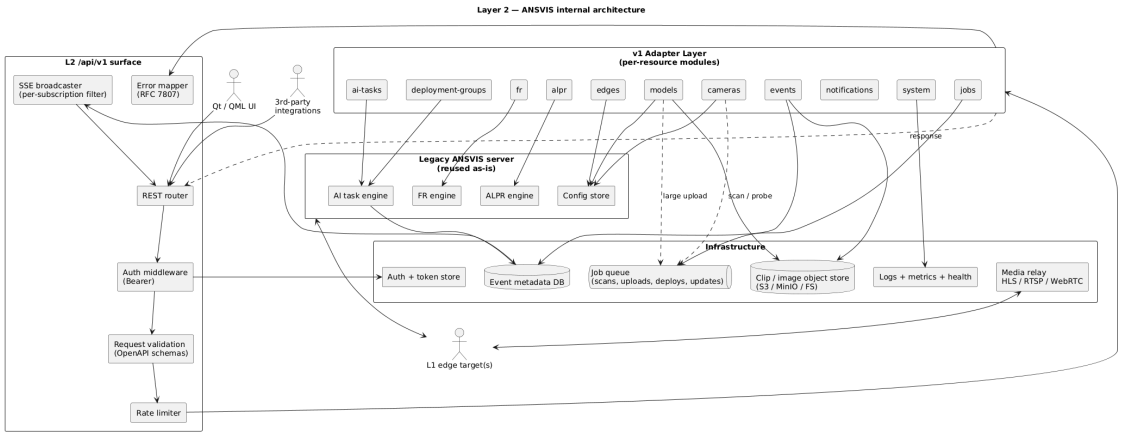
For L2 and L3 developers, L1 is treated as a remote service:

- Configuration changes go: **L3 (UI) → L2 (REST) → L1 (legacy protocol) → target reconciles**.
- Events flow: **target → L1 → L2 (legacy uplink) → L2 event store → L2 SSE → L3**.
- Liveview: **L3 asks L2 for a stream descriptor → L2 media relay pulls from L1 → L3 plays from L2's relay URL**.

The detailed L1 architecture (camera pool, inference server, frame bus, plugin SDK, etc.) is documented separately and does not affect L2 / L3 work.

# Part III — Layer 2 (ANSVIS CL)

## Architecture

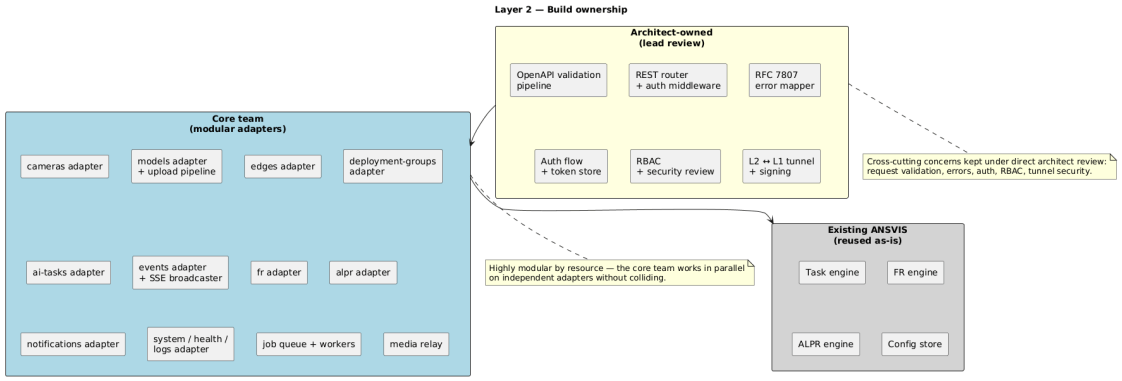



Layer 2 internals

L2 is composed of four logical groups:

1. **`/api/v1` surface** — REST router, auth middleware, OpenAPI request validation, RFC 7807 error mapper, rate limiter, SSE broadcaster.
2. **v1 adapter layer** — one module per resource (cameras, models, edges, deployment-groups, ai-tasks, events, fr, alpr, notifications, system, jobs). Each adapter translates v1 requests to one or more legacy ANSVIS calls.
3. **Legacy ANSVIS core** — reused as-is. Hosts the existing config store, AI task engine, FR engine, ALPR engine. The adapter is the only layer that calls into it.
4. **Infrastructure** — auth + token store, async job queue, event metadata DB, clip / image object store (S3 / MinIO / FS), logs + metrics, media relay.

## Ownership inside L2




L2 build ownership

- **Architect (human)** keeps direct review on the cross-cutting concerns: REST router, auth, validation pipeline, RFC 7807 errors, RBAC, L2 ⍰ L1 tunnel security.

- The **core team** builds the per-resource adapters and infrastructure helpers (job queue, media relay). Each module has a sharp boundary so the team can work in parallel without colliding.
- **Existing ANSVIS** is reused as-is. The adapter is the only entry point.

## REST API surface — `/api/v1`

The full contract is `contracts/openapi.v1.json` (OpenAPI 3.1, 67 paths, 102 operations, 91 typed schemas). Summary by resource group:

| Tag | Operations | Purpose |
|---|---|---|
| `auth` | 3 | Login, logout, refresh |
| `users` | 5 | System users CRUD |
| `cameras` | 10 | CRUD + snapshot + liveview + ONVIF scan + probe |
| `models` | 7 | CRUD + multipart upload + initiate/complete large upload |
| `edges` | 7 | CRUD + health + status |
| `deployment-groups` | 8 | CRUD + status + start/stop |
| `ai-tasks` | 11 | CRUD + status + start/stop + liveview + catalog |
| `events` | 6 | List + summary + SSE stream + image + clip + single |
| `fr` | 14 | Persons + faces + groups + detection config |
| `fr-recording` | 7 | Face-recording tasks + read/assign |
| `alpr` | 8 | Plate + plate-group CRUD |
| `notifications` | 5 | Recipients + groups |
| `triggers` | 2 | Type catalog + test dispatch |
| `system` | 7 | Status, health, info, logs, version, update |
| `jobs` | 1 | Async job polling |
| `composite` | 1 | Convenience endpoint (camera + tasks in one call) |
| **Total** | **102** | |

## Conventions enforced by `openapi.v1.json`

- Plural kebab-case resource paths ( `/cameras` , `/ai-tasks` , `/deployment-groups` ).
- `snake_case` field names. Units in the field name ( `time_interval_seconds` , not `time_interval` with `(s)` suffix).
- IDs in the path, never as query-string keys (replaces legacy `?867560018` ).
- `GET / POST / PUT / PATCH / DELETE` used semantically.
- State changes via `POST /resource/{id}/start` and `POST /resource/{id}/stop` (pragmatic exception).
- All errors as RFC 7807 problem+json with the matching HTTP status code.
- List responses use a uniform pagination wrapper: `{items, page, page_size, total, next_cursor}` .
- Auth: `Authorization: Bearer <token>` header, never in URL.
- File uploads up to 10 MB use `multipart/form-data` ; larger uploads use the initiate / complete pattern with a presigned PUT URL.
- Live events as SSE; live media via HLS / RTSP / WebRTC behind a descriptor RPC.
- All previously stringified-JSON fields ( `Parameters(json)` , `RuleConfig` , `TriggerConfig` , `ModelConfigJson` ) are now fully typed nested objects, validated against the schema.

## Migration from legacy ANSVIS endpoints

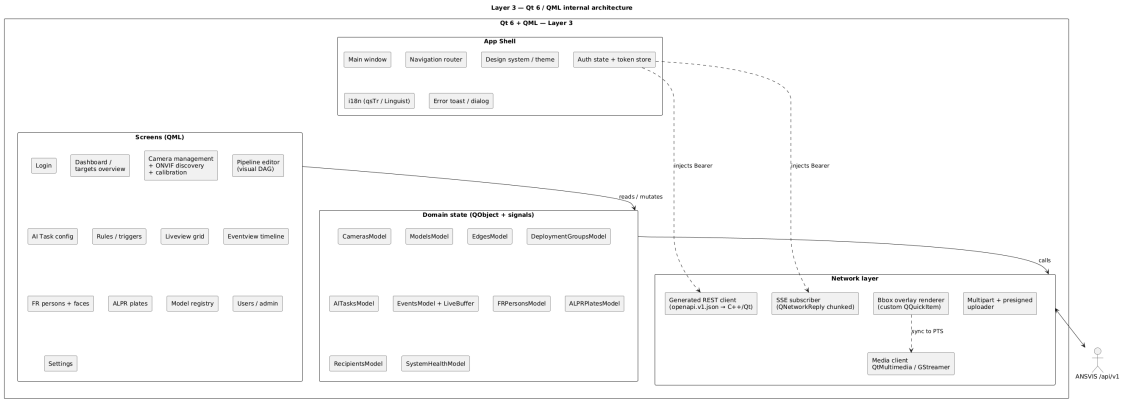
Every v1 endpoint description in the OpenAPI spec ends with `Replaces:` listing the legacy ANSVIS path(s) it supersedes. Examples:

| Legacy | v1 |
|---|---|
| `GET /Camera` (list mode) | `GET /api/v1/cameras` |
| `GET /Camera?{id}` | `GET /api/v1/cameras/{id}` |
| `DELETE /Camera?{id}` + `POST /delete/Camera?{id}` | `DELETE /api/v1/cameras/{id}` |
| `GET /ControlAITask?ID=&State=true` | `POST /api/v1/ai-tasks/{id}/start` |
| `POST /event/Get` (filter in body) | `GET /api/v1/events?task_id=&from=&to=&...` |
| `POST /FR/AddFace` (base64 in JSON) | `POST /api/v1/fr/persons/{id}/faces` (multipart) |
| `GET /SW/GetSoftwareVersion` , etc. | `GET /api/v1/system/version` , `/health` , `/info` , `/logs` , `/update` |

Legacy endpoints remain available during one release cycle with `Deprecation:` + `Sunset:` headers (RFC 8594).

# Part IV — Layer 3 (Qt / QML desktop UI)

## Architecture




Layer 3 internals

L3 is organized in four layers inside the Qt app:

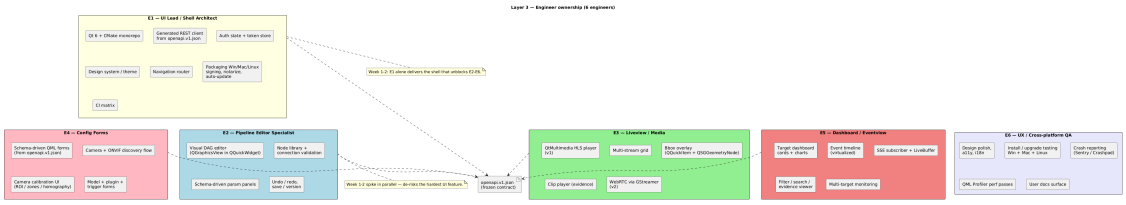
1. **App Shell** — main window, navigation router, design system / theme, i18n, auth state + token store, global error toast / dialog.
2. **Network layer** — generated REST client (from `openapi.v1.json`), SSE subscriber (chunked `QNetworkReply`), media client (QtMultimedia or GStreamer), bbox overlay renderer (custom `QQuickItem` with `QSGGeometryNode`), multipart + presigned uploader.
3. **Domain state** — `QObject` + `Q_PROPERTY` + signals models per resource (CamerasModel, ModelsModel, EdgesModel, DeploymentGroupsModel, AITasksModel, EventsModel + LiveBuffer, FRPersonsModel, ALPRPlatesModel, RecipientsModel, SystemHealthModel).
4. **Screens (QML)** — the operator surface: Login, Dashboard, Camera management, Pipeline editor, AI task config, Rules / triggers, Liveview grid, Eventview timeline, FR persons + faces, ALPR plates, Model registry, Users / admin, Settings.

## Screen → endpoint map

| Screen | Reads | Mutates | Notes |
|---|---|---|---|
| Login | — | `POST /auth/login` | Stores token in `AuthState` |
| Dashboard | `GET /edges`, `GET /edges/{id}/health`, `GET /system/health` | — | Auto-refresh every 5 s |
| Camera management | `GET /cameras`, `GET /cameras/{id}/snapshot`, `POST /cameras/probe` | `POST/PATCH/DELETE /cameras`, `POST /cameras/scan` (async job) | ONVIF discovery + calibration |
| Pipeline editor | `GET /deployment-groups/{id}`, `GET /ai-tasks/catalog`, `GET /models` | `PUT /deployment-groups/{id}` | Highest-risk feature; QGraphicsView |
| AI task config | `GET /ai-tasks/{id}`, `GET /ai-tasks/catalog` | `POST/PATCH/DELETE /ai-tasks`, `POST /ai-tasks/{id}/{start,stop}` | Schema-driven forms |
| Rules / triggers | `GET /triggers/types` | Embedded in DG / task body | `POST /triggers/test` |
| Liveview | `GET /cameras/{id}/liveview` | — | HLS v1, WebRTC v2; overlay sync to PTS |
| Eventview | `GET /events?...`, `GET /events/stream` (SSE), `/events/{id}/{image,clip}` | — | Virtualized list for 100k+ events |
| FR persons + faces | `GET /fr/persons`, `GET /fr/persons/{id}/faces` | `POST/PATCH/DELETE /fr/persons`, `POST /fr/persons/{id}/faces` (multipart) | Image upload |
| ALPR plates | `GET /alpr/plates`, `GET /alpr/plate-groups` | `POST/DELETE /alpr/plates`, `PATCH /alpr/plate-groups/{id}` | Bulk register / deregister |
| Model registry | `GET /models` | `POST /models` (multipart) or initiate / complete pattern for >100 MB | Upload progress |
| Users / admin | `GET /users` | `POST/PATCH/DELETE /users/{id}` | Admin role only |
| Settings | | | |

| Screen | Reads | Mutates | Notes |
|---|---|---|---|
| | `GET /system/info`, `GET /system/version`, `GET /system/update` | `POST /system/update` | |

## Engineer ownership




L3 engineer ownership

Six engineers, each owning a vertical slice:

| # | Role | Owns |
|---|---|---|
| **E1** | UI Lead / Shell Architect | Qt 6 + CMake monorepo, generated REST client, auth state, design system, navigation router, packaging (Win/Mac/Linux signing, notarize, auto-update), CI matrix |
| **E2** | Pipeline Editor Specialist | Visual DAG editor ( `QGraphicsView` in `QQuickWidget` ), node library, connection validation, schema-driven param panels, undo / redo, save / version |
| **E3** | Liveview / Media | QtMultimedia HLS player (v1), multi-stream grid, bbox overlay ( `QQuickItem` + `QSGGeometryNode` ), clip player, WebRTC via GStreamer (v2) |
| **E4** | Config Forms | Schema-driven QML forms from `openapi.v1.json` , camera + ONVIF discovery flow, camera calibration UI (ROI / zones / homography), model + plugin + trigger forms |
| **E5** | Dashboard / Eventview | Target dashboard cards + charts, virtualized event timeline, SSE subscriber + LiveBuffer, filter / search / evidence viewer, multi-target monitoring |
| **E6** | UX / Cross-platform QA | Design polish, accessibility, i18n, install / upgrade testing on Win / Mac / Linux, crash reporting, QML Profiler perf passes, user docs surface |

**Critical-path order for week 1–2:** E1 alone delivers the shell that unblocks E2–E6. E2 runs a node-editor spike in parallel to de-risk the hardest feature early. Once E1's shell lands, E3 / E4 / E5 work in parallel against the mock CL; E6 starts cross-platform validation.

# Part V — L2 ⍰ L3 communication

## 1. REST control plane

Every CRUD operation, every state change, every configuration push. JSON in, JSON out, RFC 7807 problem+json on error.

### Request envelope

```
GET /api/v1/cameras?page=1&page_size=50 HTTP/1.1
Host: cms.example.com
Authorization: Bearer eyJhbGc…
Accept: application/json
```

### Success list response

```
{
  "items": [
    { "id": 123, "name": "Front gate", "driver": "rtsp", "active": true, ... }
  ],
  "page": 1,
  "page_size": 50,
  "total": 247,
  "next_cursor": null
}
```

### Error response

```
{
  "type": "https://anscenter.com/errors/validation",
  "title": "Validation failed",
  "status": 422,
  "detail": "One or more fields are invalid",
  "instance": "/api/v1/cameras",
  "errors": [
    { "field": "address", "message": "Must be a valid RTSP URL" },
    { "field": "capturing.time_interval_seconds", "message": "Must be > 0" }
  ]
}
```

### Camera create — body example (clean v1)

```
{
  "name": "Front gate",
  "driver": "rtsp",
  "address": "rtsp://192.168.1.87:554/h264Preview_01_main",
  "credentials": { "username": "root", "password": "admin123" },
  "capturing": { "mode": "timer", "time_interval_seconds": 0.1 },
  "active": true,
  "logging_enabled": false,
  "edge_id": null,
  "deployment_group_ids": []
}
```

Note: no more stringified `Parameters(json)`. No more keys with parentheses or apostrophes. Credentials are nested. Units are in the field name.

## 2. SSE event stream

For live event tail:

```
GET /api/v1/events/stream?task_id=456 HTTP/1.1
Host: cms.example.com
Authorization: Bearer eyJhbGc…
Accept: text/event-stream
```

Each event:

```
event: detection
data: {"id":"20260526_181023-3","task_id":456,"camera_id":123,
       "timestamp":"2026-05-26T18:10:23Z","object":"car","score":0.91,
       "bbox":{"x":340,"y":210,"w":120,"h":80}}
```

Qt side: `QNetworkAccessManager` with chunked reads, parse newline-delimited frames, drive `EventsModel.LiveBuffer`.

## 3. Live media

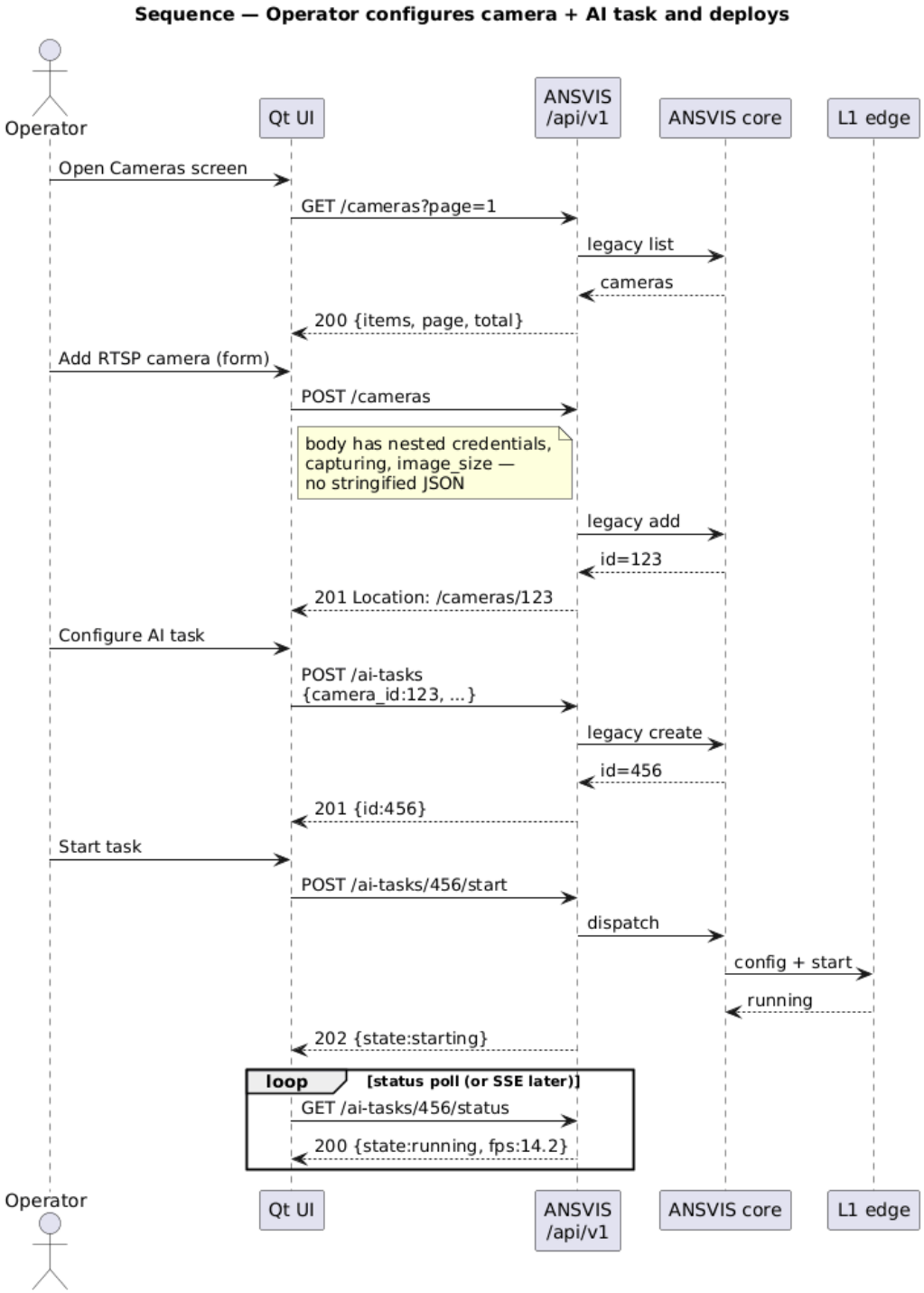
`GET /api/v1/cameras/{id}/liveview?transport=hls` returns a stream descriptor:

```
{
  "url": "https://cms.example.com/relay/cam123.m3u8",
  "transport": "hls",
  "auth_token": "tkn_short_lived_…",
  "expires_at": "2026-05-26T18:15:00Z",
  "overlay_stream": "/api/v1/events/stream?task_id=456"
}
```

Qt side: hand the URL to QtMultimedia for HLS playback (v1). For sub-second latency in v2, switch transport to `webrtc` and use GStreamer `webrtcbin`.

# Part VI — Key sequence flows

## 6.1 Operator configures a camera + AI task and deploys



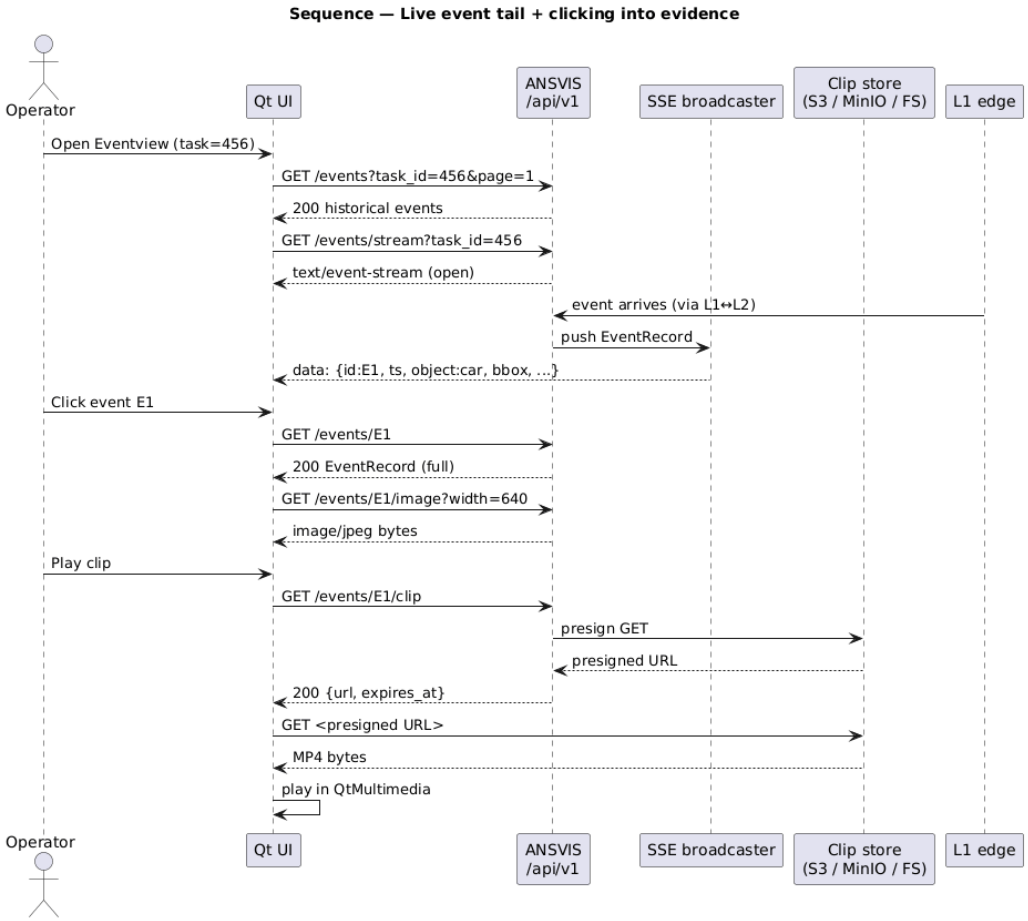

Configure & deploy sequence

Walk-through:

1. UI lists cameras via `GET /cameras?page=1`.

2. Operator submits the new-camera form. UI sends `POST /cameras` with the *clean* body (nested credentials, capturing, image_size — no stringified JSON).
3. UI creates an AI task for the new camera via `POST /ai-tasks`.
4. Operator clicks “Start”. UI calls `POST /ai-tasks/{id}/start` — returns `202 Accepted` with `{state: starting}`.
5. UI polls `GET /ai-tasks/{id}/status` until `state=running`. (Will be replaced by SSE in a later iteration so polling is optional.)

## 6.2 Live event tail with evidence

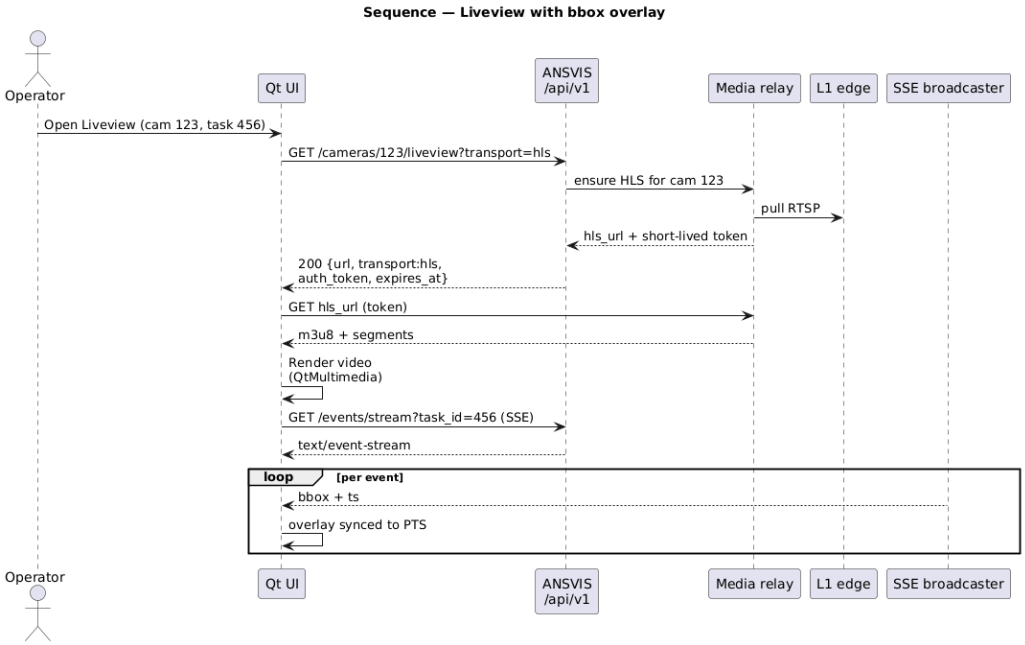



Live event tail sequence

Walk-through:

1. UI fetches historical events via `GET /events?task_id=&page=1` and renders the timeline.
2. UI opens an SSE stream `GET /events/stream?task_id=` for live tail.
3. Edge produces an event → L2 ingests → SSE broadcaster pushes the `EventRecord` to all subscribed UIs (filtered by their query).
4. Operator clicks an event. UI fetches `GET /events/{id}/image?width=640` for the snapshot.
5. Operator plays the clip. UI fetches `GET /events/{id}/clip` to get a presigned URL, then GETs the MP4 directly from the object store. The CL process never touches the video bytes — they flow client ? S3/MinIO.

## 6.3 Liveview with bbox overlay



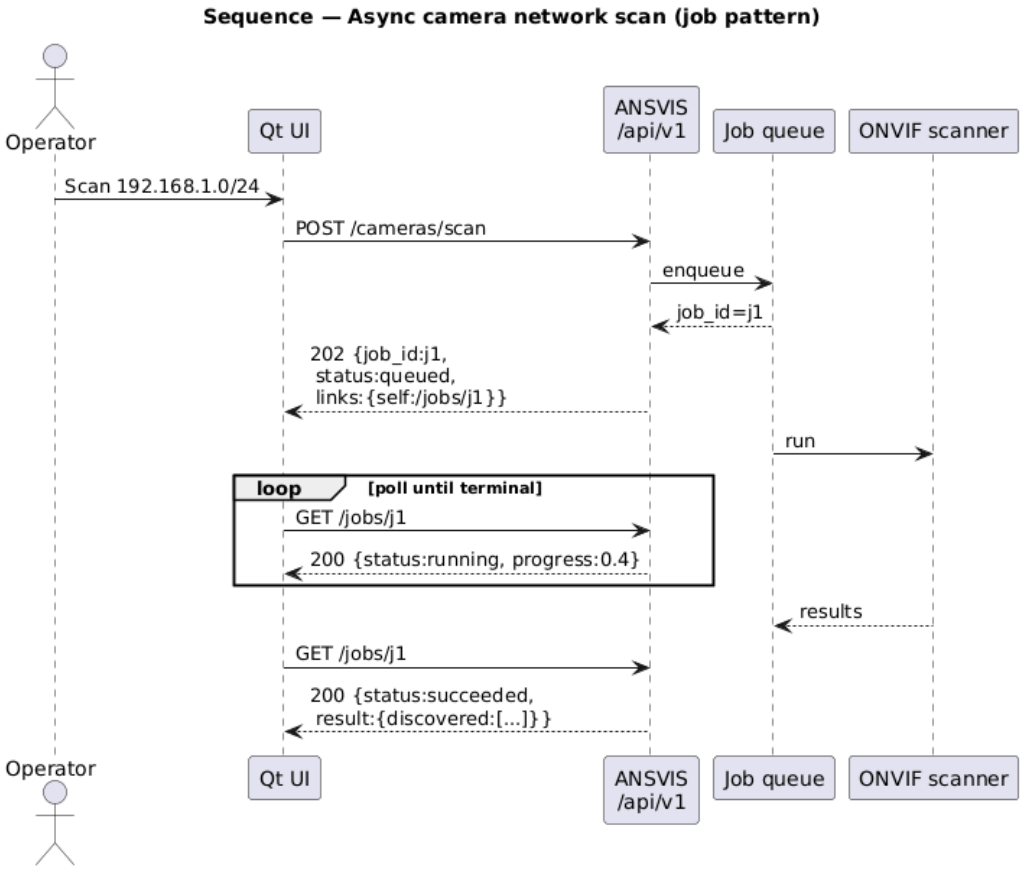

Liveview sequence

Walk-through:

1. UI requests a liveview descriptor for camera 123: `GET /cameras/123/liveview?transport=hls`.
2. L2's media relay ensures an HLS endpoint is running for that camera (pulling RTSP from the edge in the background).
3. L2 returns `{url, transport:hls, auth_token, expires_at}`.
4. UI plays the HLS URL in QtMultimedia.
5. Simultaneously the UI subscribes to `/events/stream?task_id=456` (SSE) for the AI task's detections.
6. The Qt overlay layer (custom `QQuickItem`) renders bounding boxes synced to the video PTS.

For v2 with WebRTC, the descriptor returns `transport:webrtc` and a signaling URL; GStreamer's `webrtcbin` handles negotiation.

## 6.4 Async camera scan via job queue



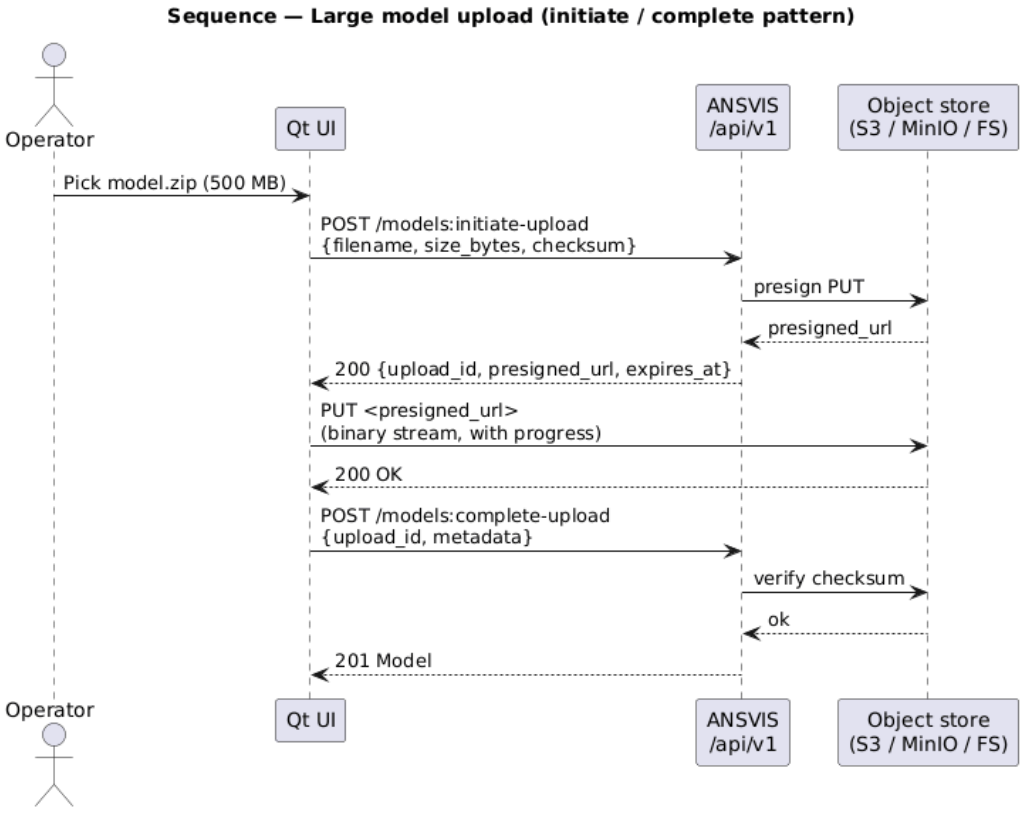

Async scan sequence

Walk-through:

1. Operator triggers an ONVIF scan of a subnet. UI calls `POST /cameras/scan` with `{ip_range, credentials, timeout_seconds}`.
2. L2 enqueues the work and immediately returns `202 Accepted` with `{job_id, status:queued, links:{self:"/jobs/j1"}}`.
3. The job worker performs the scan asynchronously.
4. UI polls `GET /jobs/{id}` until `status` becomes `succeeded` or `failed`. The terminal response includes `result` with the discovered cameras.

This pattern is used for every potentially-long operation: ONVIF scan, large model upload, deploy-to-many-targets, software update.

## 6.5 Large model upload (presigned URL)



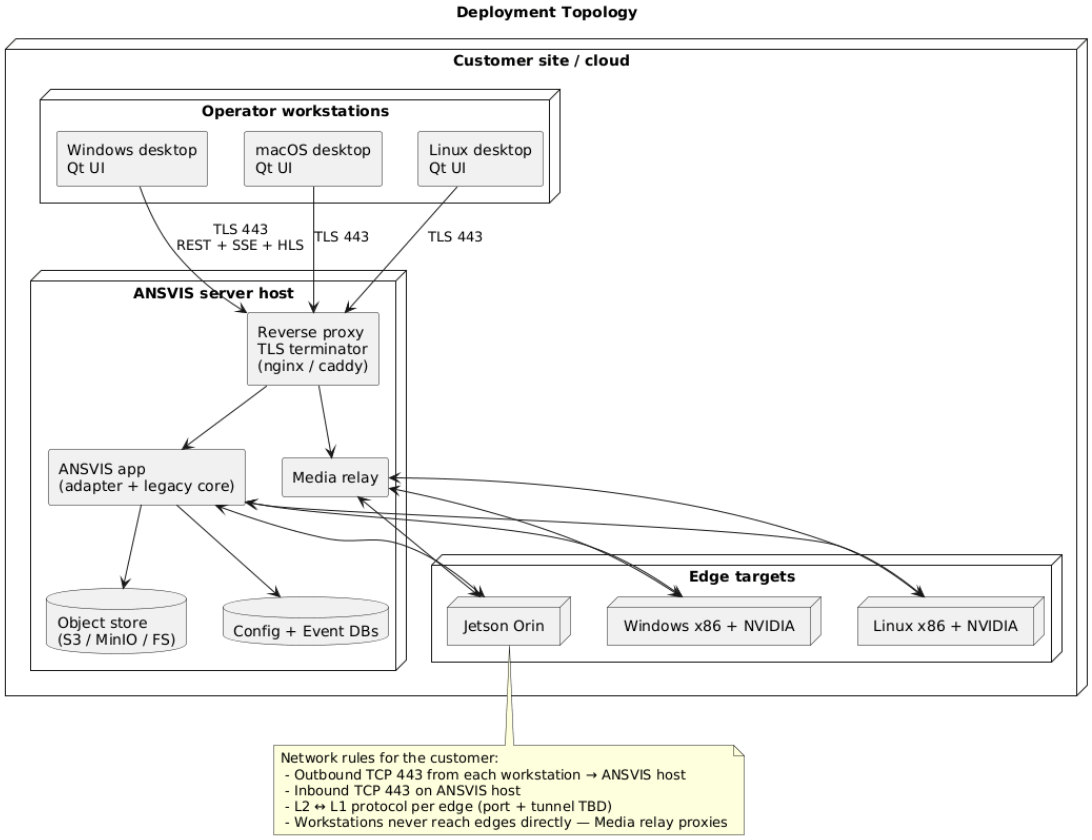

Large model upload sequence

Walk-through:

1. UI calls `POST /models:initiate-upload` with `{filename, size_bytes, checksum}`. L2 returns `{upload_id, presigned_url, expires_at}`.
2. UI PUTs the bytes directly to the object store using the presigned URL. The CL process never sees the bytes — saves CPU and avoids HTTP-body-size limits.
3. UI calls `POST /models:complete-upload` with `{upload_id, metadata}`. L2 verifies the checksum and registers the model.

For files under 10 MB, the simple `POST /models` multipart endpoint can be used instead.

# Part VII — Deployment topology




Deployment topology

## Hosts

| Host | Runs | Notes |
|---|---|---|
| Operator workstation (Win / Mac / Linux) | Qt UI application | One per concurrent operator |
| ANSVIS server host | Reverse proxy (nginx / caddy) + ANSVIS app (adapter + legacy core) + Media relay + Object store + Config / Event DBs | One per customer site (can be HA later) |
| Edge target | L1 runtime | One per physical capture site |

## Customer network rules

| Direction | Port | Purpose |
|---|---|---|
| Workstation → ANSVIS server | TCP 443 | REST + SSE + HLS over TLS |
| ANSVIS server → workstation | — (response only) | |
| ANSVIS server ? edges | TBD per deployment | Existing ANSVIS protocol |
| Workstation → edges | **None** | Workstations never reach edges directly. Media relay proxies. |

## Why workstations don’t reach edges

- Edges typically sit behind customer firewalls / NAT.
- One TLS endpoint (ANSVIS) is easier to firewall, monitor, and authenticate against than dozens of edges.
- Token rotation happens centrally.
- The media relay can transcode, cache, or split streams.

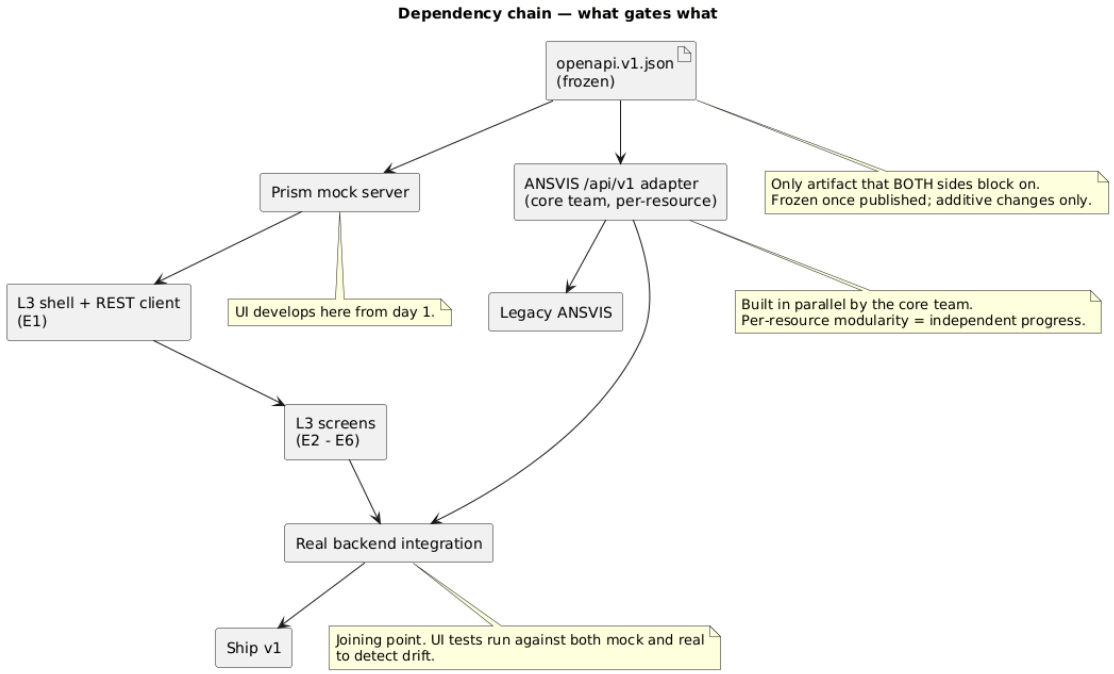
# Part VIII — Delivery plan

## Mock-first development

The contract `openapi.v1.json` is the only artifact both teams block on. Once it is published:

- **UI team** runs Stoplight Prism (`prism mock contracts/openapi.v1.json`) as a local mock CL and develops against it from day 1. No need to wait for the real adapter.
- The **backend core team** implements the adapter resource-by-resource in parallel, validating each endpoint against the same OpenAPI schema.
- **Integration** is the joining point. UI tests run against both mock and real to detect drift.

## Critical path




Dependency chain

## Sequencing

| Phase | Week | Activities |
|---|---|---|
| **Contract freeze** | 1 | Lock `openapi.v1.json` v1.0.0 (3 small decisions outstanding — see Part IX) |
| **Foundations** | 1–2 | E1: Qt 6 + CMake monorepo, generated REST client, Prism mock wired up, design system stub, app shell.<br>E2: node-editor spike (de-risk).<br>Core team: cameras / models / edges / system adapters in parallel. |
| **Vertical slices** | 3–6 | E3 (Liveview), E4 (Config forms), E5 (Dashboard / Eventview) build full features against the mock.<br>Core team: ai-tasks / deployment-groups / events / fr / alpr / notifications adapters. |
| **Real integration** | 7–8 | UI cuts over from mock to real `/api/v1` per screen as adapters land. E6 runs cross-platform install / upgrade testing on Win / Mac / Linux. |
| **Hardening** | 9–10 | Performance passes (QML Profiler, REST tracing), security review (architect-led), crash reporting, packaging signing / notarization. |
| **Ship v1** | 11–12 | Pilot deployment, customer training, feedback loop. |

# Part IX — Open decisions

These need to be locked before contract v1.0.0 publishes.

| # | Decision | Default in spec | Notes |
|---|---|---|---|
| 1 | Field naming | `snake_case` | Could switch to `camelCase` — match Qt/C++ style. One find-replace pass on `openapi.v1.json` either way. |
| 2 | Auth scheme | Bearer (existing ANSVIS token) | v1 keeps Bearer; OAuth2/OIDC for v2. |
| 3 | Mock server tool | Stoplight Prism (free) | Alternative: build a richer custom mock with scenarios. Recommend Prism. |
| 4 | Liveview transport for v1 | HLS via QtMultimedia | Easiest. Add WebRTC in v2 if sub-second latency is needed. |
| 5 | Object store | S3 / MinIO / local FS | Choose per deployment; behind a generic `ObjectStore` interface. |
| 6 | Auto-update mechanism | Sparkle (Mac), MSIX / custom (Win), AppImage (Linux) | Confirm budget for code signing per platform. |
| 7 | Contract repo location | `ANSCORE/contracts/` (current) | Separate `anscore-contracts` repo if teams diverge. |

# Appendix A — Full endpoint catalog

Complete list of v1 operations. See `contracts/openapi.v1.json` for full request / response schemas.

## auth

| Method | Path | Purpose |
|---|---|---|
| POST | `/auth/login` | Log in (returns `access_token`, `expires_at`, `user`) |
| POST | `/auth/logout` | Invalidate token |
| POST | `/auth/refresh` | Exchange refresh token for new access token |

## users

| Method | Path | Purpose |
|---|---|---|
| GET | `/users` | List users (paginated) |
| POST | `/users` | Create user |
| GET | `/users/{id}` | Get user |
| PATCH | `/users/{id}` | Update user |
| DELETE | `/users/{id}` | Delete user |

## cameras

| Method | Path | Purpose |
|---|---|---|
| GET | `/cameras` | List cameras (filters: driver, active, edge_id, deployment_group_id) |
| POST | `/cameras` | Create camera |
| GET | `/cameras/{id}` | Get camera |
| PUT | `/cameras/{id}` | Replace camera |
| PATCH | `/cameras/{id}` | Patch camera |
| DELETE | `/cameras/{id}` | Delete camera |
| GET | `/cameras/{id}/snapshot` | Get JPEG snapshot (or presigned URL) |
| GET | `/cameras/{id}/liveview` | Get liveview descriptor ( `{url, transport, token, expires_at}` ) |
| POST | `/cameras/scan` | Async ONVIF scan (returns job) |
| POST | `/cameras/probe` | Probe single IP for ONVIF profiles |

## models

| Method | Path | Purpose |
|---|---|---|
| GET | `/models` | List models (filter by type) |
| POST | `/models` | Upload + register (multipart, ≤10 MB) |
| GET | `/models/{id}` | Get model |
| PATCH | `/models/{id}` | Update model metadata |
| DELETE | `/models/{id}` | Delete model |
| POST | `/models:initiate-upload` | Begin large upload (returns presigned PUT URL) |
| POST | `/models:complete-upload` | Finalize large upload |

## edges

| Method | Path | Purpose |
|---|---|---|
| GET | `/edges` | List edges |
| POST | `/edges` | Register edge |
| GET | `/edges/{id}` | Get edge |
| PATCH | `/edges/{id}` | Update edge |
| DELETE | `/edges/{id}` | Delete edge |
| GET | `/edges/{id}/health` | Resource / temp / uptime |
| GET | `/edges/{id}/status` | State (online/offline/degraded) + counts |

## deployment-groups

| Method | Path | Purpose |
|---|---|---|
| GET | `/deployment-groups` | List groups |
| POST | `/deployment-groups` | Create group |
| GET | `/deployment-groups/{id}` | Get group |
| PUT | `/deployment-groups/{id}` | Replace group |
| DELETE | `/deployment-groups/{id}` | Delete group |
| GET | `/deployment-groups/{id}/status` | Runtime status |
| POST | `/deployment-groups/{id}/start` | Start group |
| POST | `/deployment-groups/{id}/stop` | Stop group |

## ai-tasks

| Method | Path | Purpose |
|---|---|---|
| GET | `/ai-tasks` | List tasks |
| POST | `/ai-tasks` | Create task |
| GET | `/ai-tasks/{id}` | Get task |
| PUT | `/ai-tasks/{id}` | Replace task |
| PATCH | `/ai-tasks/{id}` | Patch task |
| DELETE | `/ai-tasks/{id}` | Delete task |
| GET | `/ai-tasks/{id}/status` | Get task runtime status |
| POST | `/ai-tasks/{id}/start` | Start task |
| POST | `/ai-tasks/{id}/stop` | Stop task |
| GET | `/ai-tasks/{id}/liveview` | Liveview descriptor for annotated stream |
| GET | `/ai-tasks/catalog` | List supported task types |

## events

| Method | Path | Purpose |
|---|---|---|
| GET | `/events` | List events (filters: task_id, camera_id, object, from, to) |
| GET | `/events/summary` | Analytics rollup |
| GET | `/events/stream` | Live event tail (Server-Sent Events) |
| GET | `/events/{id}` | Get one event |
| GET | `/events/{id}/image` | JPEG snapshot |
| GET | `/events/{id}/clip` | Presigned URL for evidence clip |

## fr (face recognition)

| Method | Path | Purpose |
|---|---|---|
| GET | `/fr/persons` | List persons |
| POST | `/fr/persons` | Create person |
| GET | `/fr/persons/{id}` | Get person |
| PATCH | `/fr/persons/{id}` | Update person |
| DELETE | `/fr/persons/{id}` | Delete person |
| GET | `/fr/persons/{id}/faces` | List faces for person |
| POST | `/fr/persons/{id}/faces` | Add face (multipart, no base64) |
| DELETE | `/fr/faces/{face_id}` | Delete face |
| GET | `/fr/person-groups` | List person groups |
| POST | `/fr/person-groups` | Create group |
| PATCH | `/fr/person-groups/{id}` | Update group |
| DELETE | `/fr/person-groups/{id}` | Delete group |
| GET | `/fr/config` | Get FR detection config |
| PATCH | `/fr/config` | Update FR detection config |

## fr-recording

| Method | Path | Purpose |
|---|---|---|
| GET | `/fr/recording-tasks` | List tasks |
| POST | `/fr/recording-tasks` | Create task |
| DELETE | `/fr/recording-tasks/{id}` | Delete task |
| POST | `/fr/recording-tasks/{id}/start` | Start task |
| POST | `/fr/recording-tasks/{id}/stop` | Stop task |
| GET | `/fr/recording-tasks/{id}/faces` | Read recorded faces |
| PATCH | `/fr/recording-tasks/{id}/faces` | Assign recorded faces to a person (bulk) |

## alpr

| Method | Path | Purpose |
|---|---|---|
| GET | `/alpr/plates` | List plates |
| POST | `/alpr/plates` | Register plates (bulk) |
| DELETE | `/alpr/plates/{plate}` | Deregister single plate |
| POST | `/alpr/plates:bulk-delete` | Deregister bulk |
| GET | `/alpr/plate-groups` | List plate groups |
| POST | `/alpr/plate-groups` | Create group |
| PATCH | `/alpr/plate-groups/{id}` | Update group |
| DELETE | `/alpr/plate-groups/{id}` | Delete group |

## notifications

| Method | Path | Purpose |
|---|---|---|
| GET | `/notifications/recipients` | List recipients |
| POST | `/notifications/recipients` | Create recipient |
| PATCH | `/notifications/recipients/{id}` | Update recipient |
| DELETE | `/notifications/recipients/{id}` | Delete recipient |
| GET | `/notifications/recipient-groups` | List recipient groups |

## triggers

| Method | Path | Purpose |
|---|---|---|
| GET | `/triggers/types` | List supported trigger types + config schemas |
| POST | `/triggers/test` | Dispatch a test trigger (HTTP / MQTT / Email / etc.) |

## system

| Method | Path | Purpose |
|---|---|---|
| GET | `/system/status` | Liveness probe |
| GET | `/system/health` | Resources + counts |
| GET | `/system/info` | Hostname / OS / hardware |
| GET | `/system/logs` | Paginated logs |
| GET | `/system/version` | Version + build date + commit |
| GET | `/system/update` | Update status |
| POST | `/system/update` | Trigger update (async job) |

## jobs

| Method | Path | Purpose |
|---|---|---|
| GET | `/jobs/{id}` | Get async job status / result |

## composite

| Method | Path | Purpose |
|---|---|---|
| POST | `/composite/camera-with-tasks` | Create camera + deployment groups in one call |

# Appendix B — Artifacts in the repository

```
ANSCORE/
├── contracts/
│   └── openapi.v1.json                # The single source of truth for L2 ↔ L3
│
├── docs/
│   ├── ANSCORE-system-architecture.md  # This document (source)
│   ├── ANSCORE-system-architecture.pdf # This document (rendered)
│   ├── L2-L3-architecture.md           # Same content with Mermaid diagrams
│   └── diagrams/
│       ├── 01-system-context.puml      # PlantUML sources
│       ├── 02-l2-internals.puml
│       ├── 03-l2-ownership.puml
│       ├── 04-l3-internals.puml
│       ├── 05-seq-configure-deploy.puml
│       ├── 06-seq-live-events.puml
│       ├── 07-seq-liveview.puml
│       ├── 08-seq-async-scan.puml
│       ├── 09-seq-large-upload.puml
│       ├── 10-deployment-topology.puml
│       ├── 11-l3-engineer-ownership.puml
│       ├── 12-dependency-chain.puml
│       ├── README.md
│       ├── PREVIEW.md                  # Clickable plantuml.com URLs
│       └── png/                        # Pre-rendered PNGs (used in this PDF)
│
└── modules/                            # Existing C++ building blocks (L1 ingredients)
    ├── ANSCV/    ANSODEngine/  ANSFR/   ANSLPR/   ANSOCR/
    ├── ANSMOT/   ANSPulsar/    ANSRabbitMQ/ ANSONVIF/
    └── ...
```

# Appendix C — Glossary

| Term | Meaning |
|---|---|
| **L1** | Layer 1 — edge runtime (per target) |
| **L2** | Layer 2 — ANSVIS CL (per customer) |
| **L3** | Layer 3 — Qt/QML desktop UI (per operator workstation) |
| **Adapter (v1)** | The L2 module that translates `/api/v1` requests to legacy ANSVIS calls |
| **Edge / target** | A physical device running L1 (Windows/Linux on x86 or NVIDIA Jetson) |
| **Deployment group** | A bundle of cameras + models + rules + triggers deployed to a target |
| **AI task** | A configured pipeline (camera + model + rules + triggers) ready to run |
| **Event** | A detection emitted by an AI task, with evidence (snapshot + optional clip) |
| **Liveview** | Real-time video from a camera, with optional bbox overlay |
| **Trigger** | An outbound action when a rule fires (HTTP, MQTT, email, push, broker) |
| **ONVIF** | Camera-discovery and control protocol used by IP cameras |
| **SSE** | Server-Sent Events — one-way push from server to client over HTTPS |
| **RFC 7807** | The `application/problem+json` standard error format |
| **Prism** | Stoplight's CLI for spinning up a mock REST server from an OpenAPI spec |
| **OpenAPI 3.1** | The contract format used in `openapi.v1.json` |

*End of document.*