Architecture Decision Document

This document builds collaboratively through step-by-step discovery. Sections are appended as we work through each architectural decision together.

Project Context Analysis

Requirements Overview

Functional Requirements:
87 functional requirements across 10 capability domains. The core SA application (map, detections, sites, rules, alerts) provides the operational foundation. The AI Agent Kill Chain (14 FRs) represents the Phase 1 demo focus. Partner integration (6 FRs) and infrastructure operations (12 FRs) round out the MVP scope. The architecture must support independent development and deployment of each capability domain through Module Federation (frontend) and microservices (backend).

Non-Functional Requirements:

- **Performance:** Real-time alert delivery via WebSocket push; < 1s detection display latency; 60 FPS map rendering; 10,000+ simultaneous Cesium entities; stable memory over 8-hour shifts
- **Security:** 100% JWT-authenticated endpoints; RBAC at every service boundary with 37+ feature codes; TLS everywhere; Vault-managed secrets; OWASP Top 10 compliance; immutable audit trail
- **Scalability:** Same codebase across K3S (5-10 users), Rancher (50-100), OpenShift (1000+); horizontal scaling via stateless services; RabbitMQ handles peak sensor rates
- **Reliability:** 99.9% uptime for core SA; service isolation (no cascading failures); zero data loss during restarts; automatic K8s recovery within 60 seconds
- **Observability:** 100% distributed tracing (OpenTelemetry); Prometheus metrics on all services; centralized logs searchable within 30 seconds; Grafana dashboards
- **Maintainability:** Independent module/service deployment; protobuf backward compatibility enforced in CI; Helm-based IaC; Consul service discovery; API versioning via Tyk

Scale & Complexity:

- Primary domain: Full-stack (Cesium/React SPA + C#/Node microservices + message-driven + AI/ML agents)
- Complexity level: Enterprise / High
- Estimated architectural components: 30+
- Project context: Brownfield — modernizing a legacy C4ISR system

Technical Constraints & Dependencies

1. **WebKit base (github.com/mayafit/WebKit)** — mandatory shell application; all federated modules must integrate with this base
2. **Chromium-only browser support** — enables WebGL2/WebGPU, SharedArrayBuffer, no cross-browser polyfills needed

3. **Protobuf as canonical data model** — all internal data representation via .proto files in a dedicated Data Model repository; all services consume from this single source of truth
4. **OGC-only geospatial interface** — application must never call proprietary GeoServer APIs; WMS/WFS/WCS exclusively (exception: Phase 4 data loading module)
5. **Single-cluster deployment** — all services co-located within one K8s cluster per customer; external communication crosses cluster boundary
6. **Single-tenant per customer** — no multi-tenant logic; isolation at infrastructure level
7. **2 full-time engineers + AI augmentation** — architecture must support lean team velocity; modular boundaries enable parallel work
8. **Partner interface dependencies** — IronBrain, MSG, Omnisys, RoE Evaluator interfaces not yet fully specified; adapter pattern with mock/stub fallback required
9. **Database-per-service** — each microservice owns its data store; no shared databases
10. **Material Design (MUI)** — standard component library across all federated modules for visual consistency

Cross-Cutting Concerns Identified

Concern	Architectural Impact
JWT Authentication & Authorization	Tyk validates every inbound request; backend services validate JWT claims for defense-in-depth; frontend reads feature codes from JWT for UI gating; KeyCloak Protocol Mappers embed permissions
Protobuf Schema Governance	Central Data Model repo; backward compatibility enforced via CI; all gRPC contracts derived from shared .proto files; schema evolution must never break existing consumers
Real-Time Data Pipeline	Gateways → RabbitMQ → services → WebSocket push to clients; back-pressure handling; message persistence; queue monitoring
Observability (OpenTelemetry)	Trace ID propagation across full request chain; Prometheus metrics (RED: rate, errors, duration) on every service; structured centralized logging
Graceful Degradation	Gateway/partner failure must be isolated; core SA continues operating; mock/stub interfaces for every external dependency
Configuration & Secrets	Environment variables (externalized config); Consul for service discovery; Vault for secrets; no hardcoded addresses or credentials
Error Handling & Resilience	Configurable timeouts on all external calls; circuit breaker pattern for partner interfaces; data validation at ingestion (reject malformed, don't drop silently)
Deployment Independence	Each frontend module and backend service independently deployable via Helm; no coordinated multi-service deployments required

Frontend Layout Architecture

The Cesium 3D map is the application's primary visual element and is integral to the core frontend module — not a separate Micro Frontend. The map consumes all available viewport space after the fixed UI chrome.

Layout Structure:

- **Top Toolbar** — collapsible horizontal bar; provides access to primary navigation (tables, rules, alerts, admin, display management)
- **Map Canvas** — fills all remaining viewport space; this is the application's primary workspace; Cesium 3D globe with all entities, detections, sites, layers rendered here
- **Floating Tools Menu** — semi-transparent overlay panel; can be opened or collapsed; provides context-sensitive tools (measurement, annotations, investigation, coordinate input)
- **Status Bar** — persistent bottom bar (Windows-style); displays coordinates, sensor/system status, mission time, and other operational information

Module Federation Implication:
The shell application owns the map, toolbar, status bar, and tools menu. Federated modules (SAR exploitation, chat, AI agents, admin) inject their UI as:

- Toolbar items/buttons registered into the top toolbar
- Side panels or overlay panels that dock alongside or float over the map
- Tools menu entries registered into the floating tools menu
- Status bar segments for module-specific status information

Federated modules do NOT replace or wrap the map — they augment the map-centric layout with their domain-specific UI.

Module Federation Integration Architecture

All federated modules and the core shell are based on the **WebKit** repository (github.com/mayafit/WebKit). WebKit defines the shared runtime, integration contracts, and common libraries.

Core as Platform Model:
The core shell is not merely a layout container — it is a platform that exposes a defined API surface. Federated modules are plugins that extend the core through registration points and consume shared services.

Registration Points (Core provides, modules register into):

Registration Point	Description
Toolbar	Modules register buttons, dropdowns, and action items into the top toolbar
Tools Menu	Modules register tool entries into the floating tools menu
Status Bar	Modules register status segments into the bottom status bar

Shared Services (Core provides, modules consume):

Service	Description
Shared State	Global application state (Redux or equivalent); modules access through defined slices/actions

Service	Description
Map API	Abstraction over Cesium — draw entities, query entities, add/remove layers, control camera, subscribe to map events
Logging Interface	Structured logging API; all modules log through core's interface (not directly to console/backend)
Auth Service	JWT token access, current user context, feature permission checks
API Client	Pre-configured HTTP/gRPC client with auth headers, base URLs, error handling
WebSocket Manager	Shared WebSocket connection for real-time push (alerts, detections, chat)
Event Bus	Application-level event pub/sub for inter-module communication
Notification Service	Toast/popup notifications, alert displays

Integration Contract Stability:
Core interfaces are the contract that all modules depend on. Breaking changes to registration points or shared services break all modules. These interfaces must maintain backward compatibility or follow explicit versioning.

Starter Template Evaluation

Primary Technology Domain

Full-stack enterprise platform: Cesium/React SPA (Module Federation) + C#/Node.js microservices + message-driven architecture + AI/ML agents. Deployed on Kubernetes via Helm.

Starter: WebKit Base (github.com/mayafit/WebKit)

The starter template for Prophesee is not a generic create-app CLI — it is the **WebKit** repository, which is the mandatory foundation for the core shell and all federated modules.

Rationale: WebKit provides the Module Federation shell, React + Cesium integration baseline, and build tooling. All Prophesee frontend code (core and modules) extends from this base.

Initialization:

```
git clone https://github.com/mayafit/WebKit.git prophesee-core
```

Technology Stack (Verified Current Versions)

Frontend:

Technology	Version	Role

Technology	Version	Role
React	~19.x	UI framework
CesiumJS	~1.138	3D geospatial visualization
MUI (Material UI)	~7.x	Design system and component library
TypeScript	~5.8	Language (mandatory)
Webpack Module Federation	2.0	Microfrontend module system
Redux Toolkit	~2.11	Shared application state management

Backend:

Technology	Version	Role
.NET (C#)	10 LTS	Primary backend service language
Node.js	24 LTS	Secondary backend service language
gRPC + Protobuf	Latest stable	Service-to-service communication
RabbitMQ	~4.2	Message bus / event streaming

Data:

Technology	Version	Role
PostgreSQL + PostGIS	~18.x	Relational data, geospatial queries
MongoDB	~8.x	Document/non-relational data
Redis	~8.x	Distributed cache, shared runtime data
MinIO	Latest stable	S3-compatible object storage (status: verify — repo archived Feb 2026)

Infrastructure:

Technology	Version	Role
Kubernetes	K3S / Rancher / OpenShift	Container orchestration
Helm	~4.1	Deployment packaging
Tyk API Gateway	~5.8 LTS	API gateway, JWT auth enforcement
KeyCloak	~26.x	SSO, RBAC, identity management
Consul	~1.22	Service discovery
Vault	~1.21	Secrets management

Technology	Version	Role
GeoServer	~2.28	OGC geospatial services

Observability:

Technology	Version	Role
OpenTelemetry	SDK ~2.x	Distributed tracing, metrics
Prometheus	~3.10	Metrics collection and alerting
Grafana	~12.x	Dashboards and visualization
Elasticsearch + Beats/Logstash	Latest stable	Centralized logging

CI/CD:

Technology	Role
Git	Source control
Harbor	Container registry
ArgoCD	GitOps deployment

Architectural Decisions Established by Stack

- **Language:** TypeScript for all frontend code; C# and TypeScript/Node for backend services
- **Styling:** MUI sx/styled (Emotion-based) — inherent with MUI choice
- **State Management:** Redux Toolkit — global store owned by core; modules inject slices
- **Build Tooling:** Webpack with Module Federation 2.0 plugin
- **Data Serialization:** Protocol Buffers — single Data Model repository as source of truth
- **CSS Strategy:** MUI theme-based; deployment-specific theming via MUI theme configuration

Open Items

- **MinIO status:** Repository archived February 2026 — verify continued viability or evaluate alternative S3-compatible object storage (SeaweedFS, or cloud-native alternatives)
- **Monorepo tooling for core:** Multi-repo per architecture.MD for modules, but core + shared libraries may benefit from monorepo tooling (Nx, Turborepo, pnpm workspaces)

Testing & Quality Tooling

Testing Frameworks:

Stack	Framework	Use Case
TypeScript (Frontend + Node.js)	**Jest**	Unit tests, component tests, integration tests

Stack	Framework	Use Case
TypeScript (E2E)	**Playwright** or **Cypress** (to be decided per project needs)	End-to-end browser tests
C# (.NET)	**xUnit**	Unit tests, integration tests

Linting & Formatting:

Tool	Scope	Purpose
ESLint	All TypeScript repos	Code quality rules, import ordering, React hooks rules
Prettier	All TypeScript repos	Opinionated code formatting (semicolons, quotes, trailing commas)
EditorConfig	All repos (TypeScript + C#)	Cross-IDE consistency (indent style, line endings, charset, trim trailing whitespace)
.NET Analyzers	C# repos	Built-in Roslyn analyzers for C# code quality

Shared ESLint + Prettier configurations published from `prophesee-ci-templates` repo and consumed by all TypeScript repositories. EditorConfig file is identical across all 26 repos.

Core Architectural Decisions

Decision Priority Analysis

Critical Decisions (Block Implementation):

- Database-per-service mapping
- RabbitMQ topology and routing key convention
- WebSocket protocol (protobuf over WebSocket)
- Module Federation boundaries
- Service-to-service authentication (mTLS + JWT propagation)

Important Decisions (Shape Architecture):

- Redis caching strategy
- REST API standards
- JWT token refresh strategy
- Logging and observability standards
- Service endpoint conventions

Deferred Decisions (Post-MVP):

- MinIO replacement evaluation (RustFS)
- Monorepo tooling for core (evaluate if shared libraries grow)

Data Architecture

Database-per-Service Mapping:

Service	Database	Rationale
Detection Service	PostgreSQL + PostGIS	Geospatial queries on detection coordinates, relational integrity
Sites Service	PostgreSQL + PostGIS	Polygon geometry storage, spatial containment queries
Rule Engine Service	PostgreSQL	Structured rule presets, relational rule-to-site links
Alert Service	PostgreSQL	Relational model (alert → detection → site → rule), time-series queries
Chat Service	MongoDB	Flexible message schema, conversation threads, file attachment metadata
SAR Catalog Service	PostgreSQL + PostGIS	Image metadata with geospatial index for area-based catalog search
AI Agent Services	MongoDB	Flexible agent state, conversation history, unstructured LLM payloads
Audit Trail	MongoDB	Append-only event log, flexible schema for diverse event types
User/Auth	KeyCloak internal	KeyCloak manages its own persistence — not our concern

Redis as Distributed Cache:

Redis is used across services as a shared distributed cache layer for maintaining state and hot data:

What	Redis Pattern	Rationale
Current detection positions	Key-value with TTL	Real-time map needs fast reads; updated on every gateway message
Active alerts (open)	Sorted set by priority	Fast retrieval for alert panel; sorted by priority
User session context	Key-value with TTL	Active filters, map viewport, user preferences
Site geometry cache	Key-value	Frequently queried for rule evaluation; avoid DB round-trips
Feature permission cache	Key-value per user	Quick permission lookups derived from JWT

Object Storage:

MinIO (latest stable) for S3-compatible object storage. Interface kept S3-compatible to enable future migration to RustFS or other S3-compatible alternatives without application changes.

Authentication & Security

Service-to-Service Authentication:

- **mTLS** for all service-to-service communication within the K8s cluster — Consul Connect manages certificate rotation automatically
- **JWT propagation** for user-context calls — the originating user's JWT is forwarded through the call chain so downstream services can enforce user-level permissions
- Both mechanisms operate simultaneously: mTLS authenticates the calling service, JWT identifies the user

JWT Token Refresh Strategy:

- Short-lived access tokens (5-15 minute expiry)
- Refresh token rotation (KeyCloak native OAuth2 pattern)
- Silent refresh on the frontend before token expiry — seamless for watch floor operators on 8-hour shifts
- Refresh tokens stored in httpOnly cookies; access tokens in memory only

API & Communication Patterns

RabbitMQ Topology:

Exchange	Type	Routing Key Pattern	Purpose
`sensor.data`	Topic	`{datatype}.{sensor}`	All gateway-ingested data (e.g., `detection.capella`, `detection.sigint`, `intercept.sigint`)
`alerts`	Fanout	—	Alert notifications broadcast to all interested consumers
`system.events`	Topic	`{service}.{event}`	System-level events (health, config changes)
`agent.pipeline`	Direct	Stage name	AI agent kill chain stages (`correlation`, `roe`, `planning`, `approval`)

Routing Key Convention — datatype-first:

- Routing key is `{datatype}.{sensor}` — datatype comes first because most consumers care about the data type (e.g., all detections) regardless of which sensor produced it
- Services binding `detection.*` receive all detections from all sensors
- Services binding `detection.capella` receive only Capella SAR detections
- Multiple consumers can bind to the same pattern — RabbitMQ delivers to all bound queues
- Each service creates its own durable queue bound to the relevant exchange/routing key
- Dead letter queues for failed message processing

WebSocket Protocol:

- **Single WebSocket connection per client** managed by core's WebSocket Manager
- **Protocol Buffers over WebSocket** (binary frames) — bandwidth-efficient for high-frequency detection updates
- Protobuf message types defined in the Data Model repository alongside gRPC schemas
- Client subscribes to topics based on role and group: `detections`, `alerts`, `chat.{userId}`, `system.status`
- **WebSocket Gateway Service** on the backend subscribes to RabbitMQ, serializes to protobuf, and pushes to connected clients (filtered by user group and permissions)

REST API Standards:

- **Versioning:** URL path (`/api/v1/...`) — enforced and routed by Tyk
- **Error format:** RFC 7807 Problem Details (`{ type, title, status, detail, instance }`)
- **Pagination:** Cursor-based for large lists (detections, alerts); offset for small lists (sites, rules)
- **Request/Response format:** JSON for REST endpoints; protobuf for gRPC and WebSocket

Frontend Architecture

Module Federation Boundaries:

Module	Repository	Capability
Core Shell (prophesee-core)	Main repo	Map (Cesium), toolbar, status bar, tools menu, detection display/classify/affiliate, detection table, layout, all shared services (Redux, Map API, Auth, WebSocket, Logging, Event Bus, Notifications)
SA Module	Own repo	Sites management, rule preset management, alert management panels
Chat Module	Own repo	Chat UI, message panels, AI agent conversational interface
SAR Module	Own repo	SAR image catalog, exploitation tools (measurement, histogram, brightness/contrast, flicker comparison)
Attack Agent Module	Own repo	Planning aids UI, RoE display, traffic-light assessment, kill chain stage panels
Admin Module	Own repo	System config, GCS/platform selection, storage management, classification management

Detection management (display, classify, affiliate, table, history) lives in the **core shell** because detections are the map's primary data and fundamental to every operation.

Real-Time Data Flow:

```
Sensor Gateway → RabbitMQ → WebSocket Gateway Service → protobuf over WebSocket
→ Redux store → Cesium Map
```

- WebSocket messages (protobuf) arrive at core's WebSocket Manager
- Manager deserializes protobuf and dispatches Redux actions (`detection/added`, `detection/updated`, `alert/created`)
- Redux store updates trigger React re-renders
- Cesium entity layer reads from Redux and updates map entities
- **Entity batching:** Updates batched in 100ms windows to prevent per-entity re-renders at scale (10,000+ entities)

Infrastructure & Deployment

Logging Standard:

- **Format:** Structured JSON logs from all services
- **Mandatory fields:** `timestamp`, `level`, `service`, `traceId`, `spanId`, `message`, `userId` (if user-context), `correlationId`
- **Collection:** Beats/Logstash → Elasticsearch for storage
- **Visualization: Grafana** as the unified UI for both metrics and logs (via Elasticsearch datasource or Grafana Loki)
- Grafana provides a single pane of glass: metrics (Prometheus), logs (Elasticsearch/Loki), and traces (OpenTelemetry/Tempo)

Service Endpoint Conventions:

Every service exposes these standard endpoints:

Endpoint	Purpose	Consumer
`/health/live`	Liveness probe — is the process running?	Kubernetes
`/health/ready`	Readiness probe — can it serve traffic? (DB connected, RabbitMQ connected)	Kubernetes
`/version`	Service version, build info, git commit hash	Operations, debugging
`/metrics`	Prometheus-format metrics (request rate, latency, error rate, saturation)	Prometheus scraper

Decision Impact Analysis

Implementation Sequence:

1. Protobuf Data Model repository (schemas first — everything depends on this)
2. Core Shell with Map, Redux store, WebSocket Manager, shared services
3. RabbitMQ topology setup (exchanges, routing keys)
4. Infrastructure foundation (K8s, Helm, KeyCloak, Tyk, Consul, Vault, Redis, PostgreSQL, MongoDB, MinIO)
5. First gateway (demo data injection) → WebSocket Gateway Service → Core Shell display
6. Detection Service + Alert Service + Rule Engine Service

7. SA Module (sites, rules, alerts panels)
8. Chat Module + Correlation Agent
9. Attack Agent Module + partner integration adapters

Cross-Component Dependencies:

- All services depend on: Protobuf Data Model repo, RabbitMQ topology, Redis, JWT/KeyCloak config
- All frontend modules depend on: Core Shell shared services (Map API, Redux, Auth, WebSocket)
- WebSocket Gateway Service depends on: RabbitMQ topology + Protobuf schemas
- Rule Engine depends on: Detection Service + Sites Service data
- AI Agents depend on: Chat Module + partner adapter interfaces

Implementation Patterns & Consistency Rules

Pattern Categories Defined

Critical Conflict Points Identified:
28 areas where AI agents could make different choices — spanning naming, structure, format, communication, and process patterns. The rules below ensure all agents produce compatible, consistent code.

Naming Patterns

Database Naming Conventions:

Element	Convention	Example
PostgreSQL tables	`snake_case`, plural	`detections`, `site_rules`, `alert_histories`
PostgreSQL columns	`snake_case`	`detection_id`, `created_at`, `sensor_name`
PostgreSQL primary keys	`id` column, UUID v7	`id UUID DEFAULT gen_random_uuid()` — time-ordered for temporal data
Domain-specific IDs	Integer, named `{entity}_number`	`detection_number SERIAL` — human-readable sequential reference
Foreign keys	`{referenced_table_singular}_id`	`site_id`, `rule_id`, `detection_id`
Indexes	`idx_{table}_{columns}`	`idx_detections_sensor_name`, `idx_sites_geometry`
MongoDB collections	`camelCase`, plural	`chatMessages`, `agentStates`, `auditEvents`
MongoDB fields	`camelCase`	`userId`, `createdAt`, `messageBody`

Primary Key Strategy — UUID v7:

- All tables use UUID v7 as the primary key (`id` column) — time-ordered UUIDs that sort chronologically, enabling efficient temporal queries and distributed ID generation without coordination
- Entities also carry a **domain-specific integer ID** for human reference: `detection_number`, `site_number`, `alert_number`, `rule_number`
- Integer IDs are auto-incrementing per service database (SERIAL/SEQUENCE) — used in UI display and operator communication ("Detection #4521")
- UUID v7 is used for all foreign key references and inter-service communication
- Domain integer IDs are local to each service and never used as foreign keys across service boundaries

Protobuf Naming Conventions:

Element	Convention	Example
Package	`prophesee.{domain}.v1`	`prophesee.detection.v1`
Message	`PascalCase`	`Detection`, `SitePolygon`, `AlertEvent`
Field	`snake_case`	`sensor_name`, `detection_time`, `confidence_score`
Enum	`PascalCase` type, `UPPER_SNAKE_CASE` values	`DetectionType { DETECTION_TYPE_UNSPECIFIED = 0; DETECTION_TYPE_MOVING = 1; }`
Service	`PascalCase` + `Service` suffix	`DetectionService`, `AlertService`
RPC	`PascalCase` verb-noun	`GetDetection`, `ListAlerts`, `CreateSite`

REST API Naming Conventions:

Element	Convention	Example
Endpoints	`/api/v{n}/{resource}`, plural, kebab-case	`/api/v1/detections`, `/api/v1/site-rules`
Route parameters	`:id` (Express) / `{id}` (.NET)	`/api/v1/detections/:id`
Query parameters	`camelCase`	`?sensorName=capella&startTime=...`
Request/Response body	`camelCase` JSON fields	`{ "detectionId": "...", "sensorName": "..." }`
Headers	Standard HTTP headers; custom: `X-Prophesee-{Name}`	`X-Prophesee-Correlation-Id`

Code Naming Conventions — TypeScript (Frontend + Node.js):

Element	Convention	Example
Components	`PascalCase`	`DetectionTable`, `SiteEditor`, `AlertPanel`

Element	Convention	Example
Files (components)	`PascalCase.tsx`	`DetectionTable.tsx`, `AlertPanel.tsx`
Files (utilities)	`camelCase.ts`	`mapUtils.ts`, `dateFormatter.ts`
Files (hooks)	`use{Name}.ts`	`useDetections.ts`, `useMapEntities.ts`
Functions	`camelCase`	`getDetectionById`, `formatCoordinate`
Variables	`camelCase`	`detectionList`, `activeAlertCount`
Constants	`UPPER_SNAKE_CASE`	`MAX_ENTITY_COUNT`, `DEFAULT_MAP_ZOOM`
Interfaces/Types	`PascalCase`, no `I` prefix	`Detection`, `SiteRule`, `AlertFilter`
Enums	`PascalCase` type, `PascalCase` values	`DetectionType.Moving`, `AlertSeverity.Critical`
Redux slices	`camelCase` slice name	`detectionSlice`, `alertSlice`
Redux actions	`{slice}/{verb}`	`detection/added`, `alert/dismissed`

Code Naming Conventions — C# (.NET Backend):

Element	Convention	Example
Classes	`PascalCase`	`DetectionService`, `SiteRepository`
Files	`PascalCase.cs`	`DetectionService.cs`, `SiteRepository.cs`
Methods	`PascalCase`	`GetDetectionById`, `ProcessSensorData`
Properties	`PascalCase`	`SensorName`, `DetectionTime`
Private fields	`_camelCase`	`_detectionRepository`, `_logger`
Parameters	`camelCase`	`detectionId`, `sensorName`
Constants	`PascalCase`	`MaxRetryCount`, `DefaultTimeout`
Interfaces	`I` prefix + `PascalCase`	`IDetectionService`, `ISiteRepository`
Namespaces	`Prophesee.{Domain}.{Layer}`	`Prophesee.Detection.Services`, `Prophesee.Alert.Repositories`

Structure Patterns

Frontend Project Organization (Feature-Based):

```
prophesee-core/           # or any federated module
├── src/
│   ├── app/              # App entry, routing, providers
```

```
│   ├── features/         # Feature modules (one folder per domain)
│   │   ├── detections/
│   │   │   ├── components/
│   │   │   ├── hooks/
│   │   │   ├── slices/
│   │   │   ├── services/
│   │   │   └── types/
│   │   ├── map/
│   │   ├── alerts/
│   │   └── ...
│   ├── shared/           # Cross-feature shared code
│   │   ├── components/   # Reusable UI components
│   │   ├── hooks/        # Shared hooks
│   │   ├── services/     # API clients, WebSocket manager
│   │   ├── utils/        # Pure utility functions
│   │   └── types/        # Shared TypeScript types
│   ├── store/            # Redux store configuration
│   └── config/           # App configuration
├── tests/                # All tests in dedicated folder
│   ├── unit/
│   │   ├── features/
│   │   │   ├── detections/
│   │   │   └── alerts/
│   │   └── shared/
│   ├── integration/
│   └── e2e/
├── public/               # Static assets
└── ...config files
```

C# Backend Service Structure:

```
Prophesee.{ServiceName}/
├── src/
│   ├── Prophesee.{ServiceName}.Api/        # Web API host
│   │   ├── Controllers/
│   │   ├── Middleware/
│   │   ├── Filters/
│   │   └── Program.cs
│   ├── Prophesee.{ServiceName}.Domain/     # Domain models, interfaces
│   │   ├── Models/
│   │   ├── Interfaces/
│   │   └── Enums/
│   ├── Prophesee.{ServiceName}.Services/   # Business logic
│   │   └── ...
│   └── Prophesee.{ServiceName}.Infrastructure/  # Data access, external
│       ├── Repositories/
│       ├── Migrations/
│       └── Configuration/
├── tests/
│   ├── Prophesee.{ServiceName}.UnitTests/
│   └── Prophesee.{ServiceName}.IntegrationTests/
```

```
├── Dockerfile
└── Prophesee.{ServiceName}.sln
```

Test Location Rule:
All tests live in a dedicated `tests/` folder at the repository root — NOT co-located with source files. Test folder structure mirrors the source structure for easy navigation.

Format Patterns

API Response Formats:

Success response (single item):

```
{
  "data": { ... },
  "meta": { "requestId": "uuid-v7", "timestamp": "2026-02-28T12:00:00Z" }
}
```

Success response (collection):

```
{
  "data": [ ... ],
  "meta": { "requestId": "uuid-v7", "timestamp": "2026-02-28T12:00:00Z" },
  "pagination": { "cursor": "...", "hasMore": true, "totalCount": 1234 }
}
```

Error response (RFC 7807):

```
{
  "type": "https://prophesee.io/errors/detection-not-found",
  "title": "Detection Not Found",
  "status": 404,
  "detail": "Detection with id '01957...' does not exist",
  "instance": "/api/v1/detections/01957..."
}
```

Date/Time Format:

- All dates in JSON: ISO 8601 strings with UTC timezone — `"2026-02-28T12:00:00.000Z"`
- All dates in protobuf: `google.protobuf.Timestamp`
- All dates in PostgreSQL: `TIMESTAMPTZ` (timestamp with time zone)
- Frontend display: Localized via Intl.DateTimeFormat; operational displays use UTC with "Z" suffix

Null Handling:

- JSON: Omit null fields from responses (don't send `"field": null`)
- Protobuf: Use default values (empty string, zero, false) — proto3 semantics
- TypeScript: Use `undefined` for optional fields, never `null` (except where external APIs require it)
- C#: Use nullable reference types (`string?`, `int?`) with null checks at service boundaries

Communication Patterns

RabbitMQ Event Naming:

Pattern	Convention	Example
Routing key	`{datatype}.{sensor}`	`detection.capella`, `intercept.sigint`
Queue naming	`{service}.{purpose}`	`detection-service.ingest`, `alert-service.evaluate`
Dead letter queue	`{queue}.dlq`	`detection-service.ingest.dlq`
Exchange naming	`{domain}` (lowercase, dot-separated)	`sensor.data`, `system.events`, `alerts`

RabbitMQ Message Envelope:
Every message on RabbitMQ carries these headers:

- `messageId`: UUID v7
- `correlationId`: Trace correlation ID (from OpenTelemetry)
- `timestamp`: ISO 8601 UTC
- `source`: Producing service name
- `contentType`: `application/protobuf`
- Body: Serialized protobuf message

Redux State Management Patterns:

Pattern	Convention	Example
Slice naming	`camelCase` domain noun	`detections`, `alerts`, `sites`
Action naming	`{slice}/{pastTenseVerb}`	`detections/added`, `detections/updated`, `alerts/dismissed`
Selector naming	`select{Description}`	`selectActiveDetections`, `selectAlertsBySeverity`
Thunk naming	`{slice}/{verb}{Noun}`	`detections/fetchDetections`, `sites/createSite`
State shape	Normalized with `EntityAdapter`	`{ ids: [], entities: {}, loading: 'idle' | 'pending', error: null }`

State Update Rules:

- All state updates via Redux Toolkit (Immer-powered immutable updates)
- No direct state mutation outside of Redux slice reducers
- WebSocket messages dispatched as Redux actions by the WebSocket Manager
- Modules access state ONLY through exported selectors — never read store shape directly

Process Patterns

Error Handling:

Layer	Pattern	Example
API Controller	Catch and map to RFC 7807	`try { } catch (e) { return Problem(e); }`
Service Layer	Throw typed domain exceptions	`throw new DetectionNotFoundException(id)`
Frontend API calls	Centralized error interceptor in API Client	Auto-dispatches error notification; returns typed error
Frontend components	Error boundaries per feature area	`<DetectionErrorBoundary>` wraps detection feature
RabbitMQ consumers	Reject to DLQ after 3 retries	Exponential backoff: 1s, 5s, 30s

Logging Levels:

Level	When to Use
`ERROR`	Unrecoverable failure; requires operator attention
`WARN`	Recoverable anomaly; degraded but functional
`INFO`	Significant business events (detection received, alert created, user login)
`DEBUG`	Detailed flow for troubleshooting (disabled in production by default)

All log entries include: `timestamp`, `level`, `service`, `traceId`, `spanId`, `message`, `userId` (if available), `correlationId`.

Loading State Pattern:

```
type LoadingState = 'idle' | 'pending' | 'succeeded' | 'failed';

// In Redux slice state:
interface DetectionState {
  ids: string[];
  entities: Record<string, Detection>;
  loading: LoadingState;
  error: string | null;
}
```

- `idle`: No request made yet
- `pending`: Request in flight — show spinner/skeleton
- `succeeded`: Data loaded — render content
- `failed`: Error occurred — show error message with retry option

Validation Pattern:

- **Frontend:** Validate on form submit using schema validation (Zod or Yup); show inline field errors
- **API Gateway (Tyk):** Rate limiting, JWT validation, request size limits
- **Backend Controller:** Validate request DTO shape and required fields (FluentValidation in C#, class-validator in Node)
- **Backend Service:** Validate business rules (e.g., site polygon must be valid, detection coordinates within bounds)
- **Protobuf:** Schema enforcement at serialization/deserialization boundary

Enforcement Guidelines

All AI Agents MUST:

1. Use UUID v7 for all primary key `id` columns; add domain-specific integer IDs (`{entity}_number`) for human-facing references
2. Follow the naming conventions table exactly — no deviations (snake_case for DB/proto, camelCase for JSON/TypeScript, PascalCase for C#)
3. Place all tests in the `tests/` directory at the repository root — never co-locate tests with source files
4. Wrap all REST responses in the `{ data, meta, pagination? }` envelope; errors as RFC 7807
5. Use protobuf for all RabbitMQ messages and WebSocket frames — never JSON on these channels
6. Normalize Redux state using EntityAdapter pattern; name actions as `{slice}/{pastTenseVerb}`
7. Include all mandatory log fields (timestamp, level, service, traceId, spanId, message)
8. Handle errors at every layer: domain exceptions in services, RFC 7807 at controllers, error boundaries in React
9. Use ISO 8601 UTC for all date/time values; `TIMESTAMPTZ` in PostgreSQL
10. Follow the project structure patterns exactly — feature-based frontend, layered C# backend
11. Every backend service must expose `/health/live`, `/health/ready`, `/version`, `/metrics` endpoints

Pattern Enforcement:

- CI pipeline linting validates naming conventions
- Protobuf backward compatibility checks in CI (buf lint / buf breaking)
- Code review checklist includes pattern compliance verification
- Shared ESLint/EditorConfig rules enforced across all repositories

Pattern Examples

Good Examples:

```
// Redux action dispatch (frontend)
dispatch(detections.actions.added(newDetection));
dispatch(alerts.actions.dismissed({ alertId }));

// API endpoint (backend)
[HttpGet("/api/v1/detections/{id}")]
public async Task<IActionResult> GetDetection(Guid id)

// Database column naming
CREATE TABLE detections (
  id UUID DEFAULT gen_random_uuid() PRIMARY KEY,
  detection_number SERIAL,
  sensor_name VARCHAR(100) NOT NULL,
  detection_time TIMESTAMPTZ NOT NULL,
  confidence_score DECIMAL(5,4)
);
```

Anti-Patterns (DO NOT):

```
❌ userId instead of user_id in PostgreSQL columns
❌ /api/v1/detection (singular) instead of /api/v1/detections (plural)
❌ Placing test files next to source:
src/features/detections/DetectionTable.test.tsx
❌ JSON over WebSocket instead of protobuf binary frames
❌ { "error": "not found" } instead of RFC 7807 Problem Details
❌ Auto-increment integer as primary key instead of UUID v7
❌ Direct store.getState().detections.entities access instead of using
selectors
❌ console.log() instead of structured logging through the Logging Interface
```

Project Structure & Boundaries

Repository Landscape

The Prophesee ecosystem follows a **multi-repo** architecture. Each frontend module, backend service, sensor gateway, and partner adapter lives in its own Git repository. Shared contracts live in the Data Model repository. Deployment configuration lives in dedicated infrastructure repositories.

Total: 28 repositories

Category	Count	Repositories
Frontend Modules	6	Core Shell + 5 federated modules
Backend Services	9	Domain services + WebSocket gateway
Sensor Gateways	4	Demo injector + 3 sensor-specific (expandable)
Partner Adapters	4	IronBrain, MSG, Omnisys, RoE Evaluator

Category	Count	Repositories
Shared	2	Protobuf Data Model + CI Templates
Infrastructure	2	Helm charts + ArgoCD/K8s config
Documentation	1	Developer portal + API docs

Complete Repository Directory Structures

Frontend Repositories

1. prophesee-core (Core Shell — owns map, layout, detections, shared services)

```
prophesee-core/
├── src/
│   ├── app/                          # App entry, providers, Module Federation
host config
│   │   ├── App.tsx
│   │   ├── providers.tsx             # Redux, MUI theme, Auth, WebSocket
providers
│   │   ├── routes.tsx
│   │   └── moduleFederation.ts       # Remote module declarations
│   ├── features/
│   │   ├── map/                      # Cesium 3D map (FR1-FR15)
│   │   │   ├── components/
│   │   │   │   ├── CesiumMap.tsx
│   │   │   │   ├── EntityLayer.tsx
│   │   │   │   ├── LayerVisibilityControl.tsx
│   │   │   │   ├── CoordinateInput.tsx
│   │   │   │   ├── MeasurementTools.tsx
│   │   │   │   ├── AnnotationTools.tsx
│   │   │   │   └── TimelineControl.tsx
│   │   │   ├── hooks/
│   │   │   │   ├── useMapCamera.ts
│   │   │   │   ├── useEntitySelection.ts
│   │   │   │   └── useLayerCapabilities.ts
│   │   │   ├── slices/
│   │   │   │   ├── mapSlice.ts
│   │   │   │   └── layersSlice.ts
│   │   │   ├── services/
│   │   │   │   ├── mapApi.ts          # Map API exposed to modules
│   │   │   │   └── ogcClient.ts       # WMS/WFS/WCS requests
│   │   │   └── types/
│   │   ├── detections/               # Detection management (FR16-FR21)
│   │   │   ├── components/
│   │   │   │   ├── DetectionTable.tsx
│   │   │   │   ├── DetectionDetail.tsx
│   │   │   │   ├── DetectionClassify.tsx
│   │   │   │   ├── DetectionAffiliate.tsx
│   │   │   │   ├── DetectionHistory.tsx
```

```
│   │   │   │   └── DetectionFilters.tsx
│   │   │   ├── hooks/
│   │   │   ├── slices/
│   │   │   │   └── detectionsSlice.ts
│   │   │   ├── services/
│   │   │   └── types/
│   │   ├── layout/                  # Shell layout (toolbar, status bar,
tools menu)
│   │   │   ├── components/
│   │   │   │   ├── AppShell.tsx
│   │   │   │   ├── Toolbar.tsx
│   │   │   │   ├── StatusBar.tsx
│   │   │   │   ├── ToolsMenu.tsx
│   │   │   │   └── ModulePanel.tsx
│   │   │   ├── hooks/
│   │   │   └── slices/
│   │   │       └── layoutSlice.ts
│   │   └── filters/                 # Map entity filtering (FR11-FR12)
│   │       ├── components/
│   │       ├── hooks/
│   │       └── slices/
│   ├── shared/                      # Shared services exposed to federated
modules
│   │   ├── services/
│   │   │   ├── authService.ts       # JWT token access, permission checks
│   │   │   ├── apiClient.ts         # Pre-configured HTTP client
│   │   │   ├── webSocketManager.ts  # Protobuf WebSocket connection
│   │   │   ├── eventBus.ts          # Inter-module pub/sub
│   │   │   ├── notificationService.ts
│   │   │   └── loggingService.ts    # Structured logging interface
│   │   ├── components/              # Reusable MUI components
│   │   ├── hooks/
│   │   ├── utils/
│   │   │   ├── dateFormatter.ts
│   │   │   ├── coordinateUtils.ts
│   │   │   └── protobufDecoder.ts
│   │   └── types/
│   │       ├── moduleRegistration.ts # Toolbar/Tools/StatusBar registration
types
│   │       └── sharedContracts.ts
│   ├── store/
│   │   ├── store.ts                 # Redux store configuration
│   │   ├── rootReducer.ts
│   │   └── middleware/
│   │       └── webSocketMiddleware.ts
│   └── config/
│       ├── environment.ts
│       └── constants.ts
├── tests/
│   ├── unit/
│   │   ├── features/
│   │   │   ├── map/
│   │   │   ├── detections/
```

```
│   │   │   ├── layout/
│   │   │   └── filters/
│   │   └── shared/
│   ├── integration/
│   └── e2e/
├── public/
│   └── assets/
├── webpack.config.ts
├── tsconfig.json
├── package.json
├── .env.example
├── Dockerfile
└── README.md
```

2-6. Federated Module Template (prophesee-sa-module, prophesee-chat-module, prophesee-sar-module, prophesee-attack-agent-module, prophesee-admin-module)

All federated modules follow the same structure:

```
prophesee-{module-name}-module/
├── src/
│   ├── app/
│   │   ├── bootstrap.tsx             # Module Federation remote entry
│   │   └── moduleRegistration.ts     # Register toolbar/tools/status bar items
│   ├── features/
│   │   └── {domain}/                 # Domain-specific features
│   │       ├── components/
│   │       ├── hooks/
│   │       ├── slices/
│   │       ├── services/
│   │       └── types/
│   ├── shared/                       # Module-internal shared code
│   │   ├── components/
│   │   ├── hooks/
│   │   └── utils/
│   └── config/
├── tests/
│   ├── unit/
│   ├── integration/
│   └── e2e/
├── public/
├── webpack.config.ts
├── tsconfig.json
├── package.json
├── .env.example
├── Dockerfile
└── README.md
```

Module-specific feature folders:

Module	Feature Folders
prophesee-sa-module	`features/sites/`, `features/rules/`, `features/alerts/`
prophesee-chat-module	`features/messaging/`, `features/channels/`, `features/agentChat/`
prophesee-sar-module	`features/catalog/`, `features/exploitation/`, `features/comparison/`
prophesee-attack-agent-module	`features/killChain/`, `features/planningAids/`, `features/roeDisplay/`, `features/trafficLight/`
prophesee-admin-module	`features/systemConfig/`, `features/gcsManagement/`, `features/storageManagement/`, `features/classification/`

Backend Service Repositories (C# — .NET 10)

Template for all C# services (prophesee-detection-service, prophesee-sites-service, prophesee-rule-engine, prophesee-alert-service, prophesee-sar-catalog-service, prophesee-audit-service):

```
prophesee-{service-name}/
├── src/
│   ├── Prophesee.{ServiceName}.Api/
│   │   ├── Controllers/
│   │   │   ├── {Entity}Controller.cs
│   │   │   └── HealthController.cs    # /health/live, /health/ready, /version,
/metrics
│   │   ├── Middleware/
│   │   │   ├── CorrelationIdMiddleware.cs
│   │   │   ├── ExceptionHandlingMiddleware.cs  # RFC 7807
│   │   │   └── JwtValidationMiddleware.cs
│   │   ├── Filters/
│   │   ├── Dto/                       # Request/Response DTOs
│   │   └── Program.cs
│   ├── Prophesee.{ServiceName}.Domain/
│   │   ├── Models/
│   │   ├── Interfaces/
│   │   │   ├── I{Entity}Repository.cs
│   │   │   └── I{Entity}Service.cs
│   │   ├── Enums/
│   │   └── Exceptions/                # Typed domain exceptions
│   ├── Prophesee.{ServiceName}.Services/
│   │   └── {Entity}Service.cs
│   └── Prophesee.{ServiceName}.Infrastructure/
│       ├── Repositories/
│       │   └── {Entity}Repository.cs
│       ├── Migrations/
│       ├── Consumers/                  # RabbitMQ consumers
│       │   └── {Event}Consumer.cs
```

```
│       ├── Configuration/
│       │   ├── DatabaseConfiguration.cs
│       │   ├── RabbitMqConfiguration.cs
│       │   └── ConsulRegistration.cs
│       └── Telemetry/
│           └── OpenTelemetrySetup.cs
├── tests/
│   ├── Prophesee.{ServiceName}.UnitTests/
│   │   ├── Services/
│   │   └── Controllers/
│   └── Prophesee.{ServiceName}.IntegrationTests/
├── Dockerfile
├── Prophesee.{ServiceName}.sln
└── README.md
```

Backend Service Repositories (Node.js)

prophesee-websocket-gateway and prophesee-chat-service use Node.js:

```
prophesee-{service-name}/
├── src/
│   ├── app.ts                        # Entry point
│   ├── config/
│   │   ├── environment.ts
│   │   ├── rabbitmq.ts
│   │   └── redis.ts
│   ├── controllers/                   # REST endpoints (if any)
│   │   └── healthController.ts       # /health/live, /health/ready, /version,
/metrics
│   ├── services/
│   │   └── {domain}Service.ts
│   ├── consumers/                     # RabbitMQ consumers
│   │   └── {event}Consumer.ts
│   ├── middleware/
│   │   ├── correlationId.ts
│   │   ├── errorHandler.ts           # RFC 7807
│   │   └── jwtValidation.ts
│   ├── types/
│   └── utils/
│       ├── logger.ts                  # Structured logging
│       └── telemetry.ts               # OpenTelemetry setup
├── tests/
│   ├── unit/
│   └── integration/
├── Dockerfile
├── tsconfig.json
├── package.json
└── README.md
```

prophesee-agent-service (Node.js — LLM/AI agent orchestration):

```
prophesee-agent-service/
├── src/
│   ├── app.ts
│   ├── config/
│   ├── agents/
│   │   ├── correlationAgent/         # FR43-FR44
│   │   │   ├── correlationAgent.ts
│   │   │   ├── prompts/
│   │   │   └── tools/
│   │   └── planningApprovalAgent/    # FR50
│   │       ├── planningApprovalAgent.ts
│   │       ├── prompts/
│   │       └── tools/
│   ├── controllers/
│   │   ├── agentController.ts
│   │   └── healthController.ts
│   ├── services/
│   │   ├── agentOrchestrator.ts
│   │   └── hitlGateService.ts        # FR54 — HITL approval gates
│   ├── consumers/
│   ├── middleware/
│   ├── types/
│   └── utils/
├── tests/
│   ├── unit/
│   └── integration/
├── Dockerfile
├── tsconfig.json
├── package.json
└── README.md
```

Sensor Gateway Repositories

Each sensor gateway follows the same pattern:

```
prophesee-gateway-{sensor}/
├── src/
│   ├── app.ts                        # (or Program.cs)
│   ├── config/
│   ├── ingest/
│   │   ├── {sensor}Listener.ts       # Receives raw sensor data
│   │   └── {sensor}Parser.ts         # Parses to canonical protobuf
│   ├── publishers/
│   │   └── rabbitmqPublisher.ts      # Publishes to sensor.data exchange
│   ├── controllers/
│   │   └── healthController.ts
│   ├── types/
```

```
│   └── utils/
├── tests/
│   ├── unit/
│   └── integration/
├── Dockerfile
├── package.json (or .sln)
└── README.md
```

Specific gateways:

Repository	Sensor	Phase	Notes
prophesee-gateway-demo-injector	JSON demo data	Phase 1	FR41 — simulates SAR + SIGINT feeds
prophesee-gateway-capella	Capella SAR	Phase 2	Live satellite imagery
prophesee-gateway-ais	AIS maritime	Phase 2	AIS transponder data
prophesee-gateway-sigint	SIGINT	Phase 3	Signals intelligence

Partner Adapter Repositories

Each partner adapter follows the adapter pattern:

```
prophesee-adapter-{partner}/
├── src/
│   ├── app.ts                      # (or Program.cs)
│   ├── config/
│   ├── adapter/
│   │   ├── {partner}Client.ts      # External partner API client
│   │   ├── {partner}Mapper.ts      # Maps between partner format and
protobuf
│   │   └── {partner}MockStub.ts    # FR87 — mock/stub fallback
│   ├── consumers/
│   │   └── {trigger}Consumer.ts    # RabbitMQ consumer for triggering
partner calls
│   ├── publishers/
│   │   └── resultPublisher.ts      # Publishes results back to RabbitMQ
│   ├── controllers/
│   │   └── healthController.ts
│   ├── types/
│   └── utils/
├── tests/
│   ├── unit/
│   └── integration/
├── Dockerfile
```

```
├── package.json (or .sln)
└── README.md
```

Specific adapters:

Repository	Partner	Interface	Phase
prophesee-adapter-ironbrain	IronBrain	SAR ATC results (FR42, FR82)	Phase 1
prophesee-adapter-msg	MSG	Weaponeering (FR48-FR49, FR83)	Phase 1
prophesee-adapter-omnisys	Omnisys	Tasking/BDA (FR52-FR53, FR84)	Phase 1
prophesee-adapter-roe-evaluator	RoE Evaluator	RoE assessment (FR45, FR85)	Phase 1

Shared Repositories

prophesee-data-model (Protobuf schemas — single source of truth):

```
prophesee-data-model/
├── proto/
│   ├── prophesee/
│   │   ├── common/
│   │   │   └── v1/
│   │   │       ├── common.proto       # Shared types (Coordinate, TimeRange,
Pagination)
│   │   │       └── enums.proto        # Shared enums
│   │   ├── detection/
│   │   │   └── v1/
│   │   │       ├── detection.proto    # Detection message, DetectionType enum
│   │   │       └── detection_service.proto  # gRPC service definition
│   │   ├── site/
│   │   │   └── v1/
│   │   │       ├── site.proto
│   │   │       └── site_service.proto
│   │   ├── alert/
│   │   │   └── v1/
│   │   │       ├── alert.proto
│   │   │       └── alert_service.proto
│   │   ├── chat/
│   │   │   └── v1/
│   │   │       └── chat.proto
│   │   ├── agent/
│   │   │   └── v1/
│   │   │       ├── agent.proto        # Agent messages, kill chain stages
│   │   │       └── planning_aid.proto
│   │   ├── sar/
│   │   │   └── v1/
│   │   │       └── sar_catalog.proto
│   │   ├── websocket/
```

```
│   │   │   └── v1/
│   │   │       └── push_message.proto # WebSocket push message wrapper
│   │   └── audit/
│   │       └── v1/
│   │           └── audit_event.proto
├── buf.yaml                          # Buf configuration
├── buf.gen.yaml                      # Code generation config
├── buf.lock
└── README.md
```

prophesee-helm-charts:

```
prophesee-helm-charts/
├── charts/
│   ├── prophesee-core/
│   ├── prophesee-sa-module/
│   ├── prophesee-chat-module/
│   ├── prophesee-sar-module/
│   ├── prophesee-attack-agent-module/
│   ├── prophesee-admin-module/
│   ├── prophesee-detection-service/
│   ├── prophesee-sites-service/
│   ├── prophesee-rule-engine/
│   ├── prophesee-alert-service/
│   ├── prophesee-chat-service/
│   ├── prophesee-sar-catalog-service/
│   ├── prophesee-agent-service/
│   ├── prophesee-audit-service/
│   ├── prophesee-websocket-gateway/
│   ├── prophesee-gateway-demo-injector/
│   ├── prophesee-gateway-capella/
│   ├── prophesee-adapter-ironbrain/
│   ├── prophesee-adapter-msg/
│   ├── prophesee-adapter-omnisys/
│   ├── prophesee-adapter-roe-evaluator/
│   └── prophesee-umbrella/           # Umbrella chart for full deployment
│       ├── Chart.yaml
│       ├── values.yaml               # Default deployment profile
│       ├── values-tactical.yaml      # K3S: 5-10 users
│       ├── values-theater.yaml       # Rancher: 50-100 users
│       └── values-strategic.yaml     # OpenShift: 1000+ users
├── README.md
└── DEPLOYMENT.md
```

prophesee-infrastructure:

```
prophesee-infrastructure/
├── argocd/
```

```
│   ├── applications/                # ArgoCD Application CRDs per service
│   └── projects/
├── k8s/
│   ├── namespaces/
│   ├── rbac/
│   ├── network-policies/
│   └── storage-classes/
├── platform/
│   ├── keycloak/                    # Realm config, protocol mappers, roles
│   ├── tyk/                         # API gateway config, rate limits,
policies
│   ├── consul/                      # Service mesh config
│   ├── vault/                       # Secrets engine config, policies
│   ├── rabbitmq/                    # Exchange/queue definitions, policies
│   ├── postgresql/                  # Database init scripts per service
│   ├── mongodb/                     # Database init scripts
│   ├── redis/                       # Redis cluster config
│   ├── minio/                       # Bucket definitions, policies
│   ├── geoserver/                   # OGC service config, data directory
│   ├── prometheus/                  # Alerting rules, scrape configs
│   ├── grafana/                     # Dashboard provisioning, datasources
│   └── elasticsearch/               # Index templates, ILM policies
└── README.md
```

prophesee-ci-templates (Shared CI/CD pipelines, linting configs, and quality tooling):

```
prophesee-ci-templates/
├── pipelines/
│   ├── typescript-build.yml         # Reusable build pipeline for TS repos
│   ├── dotnet-build.yml             # Reusable build pipeline for C# repos
│   ├── docker-build-push.yml        # Build + push to Harbor
│   ├── helm-lint-test.yml           # Helm chart validation
│   └── protobuf-lint-breaking.yml   # Buf lint + breaking change detection
├── configs/
│   ├── eslint/
│   │   ├── base.js                  # Base ESLint config for all TS repos
│   │   ├── react.js                 # React-specific extensions
│   │   └── node.js                  # Node.js-specific extensions
│   ├── prettier/
│   │   └── .prettierrc.json         # Shared Prettier config
│   ├── typescript/
│   │   ├── tsconfig.base.json       # Base TypeScript config
│   │   ├── tsconfig.react.json      # React-specific TS config
│   │   └── tsconfig.node.json       # Node.js-specific TS config
│   ├── jest/
│   │   ├── jest.config.base.ts      # Base Jest config
│   │   ├── jest.config.react.ts     # React testing config
│   │   └── jest.config.node.ts      # Node.js testing config
│   ├── .editorconfig                # Universal EditorConfig for all repos
│   └── .gitignore                   # Shared .gitignore template
├── scripts/
```

```
│   ├── version-check.sh              # Version consistency validation
│   └── protobuf-compat-check.sh      # Protobuf backward compatibility
└── README.md
```

prophesee-docs (Developer portal, API documentation, architecture reference):

```
prophesee-docs/
├── docs/
│   ├── architecture/
│   │   ├── overview.md               # Architecture summary and diagrams
│   │   ├── data-flow.md              # End-to-end data flow documentation
│   │   ├── module-federation.md      # Module integration guide
│   │   └── security.md               # Security architecture reference
│   ├── api/
│   │   ├── rest/                     # Generated OpenAPI/Swagger specs per
service
│   │   │   ├── detection-service.yaml
│   │   │   ├── sites-service.yaml
│   │   │   ├── alert-service.yaml
│   │   │   └── ...
│   │   ├── grpc/                     # Generated gRPC service docs from proto
│   │   ├── websocket/                # WebSocket message catalog
│   │   └── rabbitmq/                 # Exchange/queue/routing documentation
│   ├── guides/
│   │   ├── getting-started.md        # Developer onboarding
│   │   ├── creating-a-module.md      # How to create a new federated module
│   │   ├── creating-a-gateway.md     # How to create a new sensor gateway
│   │   ├── creating-an-adapter.md    # How to create a new partner adapter
│   │   └── creating-a-service.md     # How to create a new backend service
│   ├── operations/
│   │   ├── deployment.md             # Deployment procedures
│   │   ├── monitoring.md             # Grafana dashboards and alerts guide
│   │   └── troubleshooting.md        # Common issues and resolution
│   └── standards/
│       ├── naming-conventions.md     # Extracted from architecture doc
│       ├── api-standards.md          # REST/gRPC/WebSocket standards
│       └── testing-standards.md      # Testing requirements and patterns
├── static/
│   ├── diagrams/                     # Architecture diagrams (draw.io,
Mermaid)
│   └── images/
├── docusaurus.config.js              # (or similar static site generator)
├── package.json
└── README.md
```

Requirements Traceability Matrix (498-Style)

FR → Repository/Service Mapping

9.1 Map & Geospatial Visualization (FR1-FR15)

FR	Description	Frontend Repo	Backend Service	Database
FR1	3D globe with Cesium	prophesee-core (map/)	—	—
FR2	Map navigation (pan, zoom, rotate)	prophesee-core (map/)	—	—
FR3	Detection entities on map	prophesee-core (map/, detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR4	Detection metadata detail panel	prophesee-core (detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR5	Entity movement trails	prophesee-core (map/)	prophesee-detection-service	PostgreSQL+PostGIS
FR6	Coordinate input / jump-to	prophesee-core (map/)	—	—
FR7	Coordinate system switching	prophesee-core (map/)	—	—
FR8	Measurement tools	prophesee-core (map/)	—	—
FR9	Map annotations	prophesee-core (map/)	prophesee-detection-service	PostgreSQL+PostGIS
FR10	Historical data replay / timeline	prophesee-core (map/)	prophesee-detection-service	PostgreSQL+PostGIS
FR11	Entity filter by type	prophesee-core (filters/)	—	—
FR12	Entity filter by geographic area	prophesee-core (filters/)	—	—
FR13	Layer visibility control (WMS)	prophesee-core (map/)	GeoServer (OGC)	—
FR14	Show/hide layers	prophesee-core (map/)	GeoServer (OGC)	—
FR15	Layer z-order reordering	prophesee-core (map/)	—	—

9.2 Detection Management (FR16-FR21)

FR	Description	Frontend Repo	Backend Service	Database

FR	Description	Frontend Repo	Backend Service	Database
FR16	Map + table views	prophesee-core (detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR17	Classify detections	prophesee-core (detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR18	Affiliate detections	prophesee-core (detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR19	Detection history / change log	prophesee-core (detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR20	Detection filtering	prophesee-core (detections/)	prophesee-detection-service	PostgreSQL+PostGIS
FR21	Real-time detection ingestion	prophesee-core (detections/)	prophesee-detection-service, prophesee-websocket-gateway	PostgreSQL+PostGIS, Redis

9.3 Sites Management (FR22-FR26)

FR	Description	Frontend Repo	Backend Service	Database
FR22	CRUD polygon sites	prophesee-sa-module (sites/)	prophesee-sites-service	PostgreSQL+PostGIS
FR23	Site parameters/metadata	prophesee-sa-module (sites/)	prophesee-sites-service	PostgreSQL+PostGIS
FR24	Link rule presets to sites	prophesee-sa-module (sites/)	prophesee-sites-service, prophesee-rule-engine	PostgreSQL
FR25	Sites on map (styled)	prophesee-core (map/) + prophesee-sa-module	prophesee-sites-service	PostgreSQL+PostGIS
FR26	Site priority levels	prophesee-sa-module (sites/)	prophesee-sites-service	PostgreSQL+PostGIS

9.4 Rule Engine & Alerting (FR27-FR34)

FR	Description	Frontend Repo	Backend Service	Database
FR27	CRUD rule presets	prophesee-sa-module (rules/)	prophesee-rule-engine	PostgreSQL

FR	Description	Frontend Repo	Backend Service	Database
FR28	Assign rules to sites	prophesee-sa-module (rules/)	prophesee-rule-engine, prophesee-sites-service	PostgreSQL
FR29	Automatic rule evaluation	—	prophesee-rule-engine	PostgreSQL
FR30	Alert panel (priority-ordered)	prophesee-sa-module (alerts/)	prophesee-alert-service	PostgreSQL
FR31	Acknowledge/escalate alerts	prophesee-sa-module (alerts/)	prophesee-alert-service	PostgreSQL
FR32	Investigate alerts (drill down)	prophesee-sa-module (alerts/)	prophesee-alert-service, prophesee-detection-service	PostgreSQL
FR33	Commander review escalated	prophesee-sa-module (alerts/)	prophesee-alert-service	PostgreSQL
FR34	Real-time alert push (WebSocket)	prophesee-core (shared/webSocketManager)	prophesee-alert-service, prophesee-websocket-gateway	PostgreSQL, Redis

9.5 Collaborative Chat (FR35-FR40)

FR	Description	Frontend Repo	Backend Service	Database
FR35	1-to-1 messaging	prophesee-chat-module (messaging/)	prophesee-chat-service	MongoDB
FR36	Group/channel messaging	prophesee-chat-module (channels/)	prophesee-chat-service	MongoDB
FR37	Share data items in chat	prophesee-chat-module (messaging/)	prophesee-chat-service	MongoDB
FR38	Share map pointers in chat	prophesee-chat-module (messaging/)	prophesee-chat-service	MongoDB
FR39	File attachments	prophesee-chat-module (messaging/)	prophesee-chat-service	MongoDB, MinIO
FR40	AI agent chat interface	prophesee-chat-module (agentChat/)	prophesee-agent-service, prophesee-chat-service	MongoDB

9.6 AI Agent Kill Chain (FR41-FR54)

FR	Description	Frontend Repo	Backend Service	Adapter/Gateway

FR	Description	Frontend Repo	Backend Service	Adapter/Gateway
FR41	Demo data injection	—	prophesee-gateway-demo-injector	—
FR42	IronBrain ATC results	prophesee-core (detections/)	prophesee-detection-service	prophesee-adapter-ironbrain
FR43	Correlation Agent query	prophesee-chat-module (agentChat/)	prophesee-agent-service	—
FR44	Correlated entity display	prophesee-core (map/)	prophesee-agent-service	—
FR45	Send to RoE evaluator	—	prophesee-agent-service	prophesee-adapter-roe-evaluator
FR46	RoE results as alerts	prophesee-attack-agent-module (roeDisplay/)	prophesee-alert-service	prophesee-adapter-roe-evaluator
FR47	Create planning aids	prophesee-attack-agent-module (planningAids/)	prophesee-agent-service	—
FR48	Approve & send to MSG	prophesee-attack-agent-module (planningAids/)	prophesee-agent-service	prophesee-adapter-msg
FR49	Receive MSG strike plans	prophesee-attack-agent-module (killChain/)	prophesee-agent-service	prophesee-adapter-msg
FR50	Traffic-light assessment	prophesee-attack-agent-module (trafficLight/)	prophesee-agent-service	—
FR51	Commander HITL approval	prophesee-attack-agent-module (trafficLight/)	prophesee-agent-service	—
FR52	Send to Omnisys	—	prophesee-agent-service	prophesee-adapter-omnisys
FR53	Receive BDA plans	prophesee-attack-agent-module (killChain/)	prophesee-agent-service	prophesee-adapter-omnisys
FR54	HITL gates at every stage	prophesee-attack-agent-module	prophesee-agent-service	—

9.7 SAR Exploitation (FR55-FR60)

FR	Description	Frontend Repo	Backend Service	Database

FR	Description	Frontend Repo	Backend Service	Database
FR55	SAR catalog search	prophesee-sar-module (catalog/)	prophesee-sar-catalog-service	PostgreSQL+PostGIS
FR56	SAR image overlay on map	prophesee-sar-module (exploitation/) + prophesee-core (map/)	prophesee-sar-catalog-service	MinIO
FR57	Brightness/contrast adjust	prophesee-sar-module (exploitation/)	—	—
FR58	Measurement tools on SAR	prophesee-sar-module (exploitation/)	—	—
FR59	Flicker comparison	prophesee-sar-module (comparison/)	prophesee-sar-catalog-service	MinIO
FR60	SAR histogram	prophesee-sar-module (exploitation/)	—	—

9.8 Identity, Access & Data Scoping (FR61-FR69)

FR	Description	Frontend Repo	Backend Service	Infrastructure
FR61	KeyCloak SSO login	prophesee-core (shared/authService)	KeyCloak	KeyCloak
FR62	Tyk JWT enforcement	—	Tyk API Gateway	Tyk
FR63	Feature codes per role	prophesee-core (shared/authService)	KeyCloak	KeyCloak
FR64	Frontend permission gating	prophesee-core (shared/authService)	KeyCloak, Tyk	KeyCloak
FR65	Group-based data scoping (Ph4)	—	All data services	—
FR66	Ungrouped = see all (Ph4)	—	All data services	—
FR67	Multi-group membership (Ph4)	—	KeyCloak	KeyCloak
FR68	User/role/group admin	prophesee-admin-module	KeyCloak	KeyCloak
FR69	Immutable audit trail	—	prophesee-audit-service	MongoDB

9.9 Platform Infrastructure & Operations (FR70-FR81)

FR	Description	Repository	Infrastructure Component
FR70	Helm-based K8s deployment	prophesee-helm-charts	Kubernetes
FR71	Independent module deployment	prophesee-helm-charts	ArgoCD
FR72	Gateway connection config	prophesee-admin-module + prophesee-infrastructure	Consul
FR73	GeoServer layer management	prophesee-admin-module	GeoServer
FR74	Prometheus/Grafana dashboards	prophesee-infrastructure (grafana/)	Prometheus, Grafana
FR75	Distributed tracing	All services (telemetry/)	OpenTelemetry
FR76	Centralized log viewing	prophesee-infrastructure (elasticsearch/)	Grafana, Elasticsearch
FR77	RabbitMQ monitoring	prophesee-infrastructure	RabbitMQ Management
FR78	System status to watch floor	prophesee-core (layout/StatusBar)	prophesee-websocket-gateway
FR79	Graceful degradation	All services	Kubernetes, Circuit breaker
FR80	Vault secrets management	prophesee-infrastructure (vault/)	Vault
FR81	MinIO object storage	All services needing files	MinIO

9.10 Partner System Integration (FR82-FR87)

FR	Description	Adapter Repository	Pattern
FR82	IronBrain interface	prophesee-adapter-ironbrain	Adapter + interface contract
FR83	MSG interface	prophesee-adapter-msg	Adapter + interface contract
FR84	Omnisys interface	prophesee-adapter-omnisys	Adapter + interface contract
FR85	RoE evaluator interface	prophesee-adapter-roe-evaluator	Adapter + interface contract
FR86	Adapter isolation pattern	All adapters	Enforced by separate repos
FR87	Mock/stub fallback	All adapters ({partner}MockStub)	Configurable via env var

Architectural Boundaries

API Boundaries:

Boundary	Protocol	Auth	Direction
Browser → Tyk Gateway	HTTPS (REST)	JWT (validated by Tyk)	Client → Gateway
Tyk → Backend Services	HTTP/gRPC	JWT propagation + mTLS	Gateway → Service
Service → Service	gRPC	mTLS + JWT propagation	Service → Service
Service → RabbitMQ	AMQP	mTLS	Service → Message Bus
Service → Database	TCP (native)	mTLS + credentials (Vault)	Service → Data
WebSocket Gateway → Browser	WSS (protobuf)	JWT (connection auth)	Service → Client
Gateway → External Sensor	Sensor-specific	Sensor-specific	Inbound
Adapter → Partner System	Partner-specific	Partner-specific	Outbound

Data Boundaries (Database-per-Service):

Service	Database	Owns
Detection Service	PostgreSQL+PostGIS	`detections`, `detection_history`, `annotations`
Sites Service	PostgreSQL+PostGIS	`sites`, `site_metadata`
Rule Engine	PostgreSQL	`rule_presets`, `rule_assignments`
Alert Service	PostgreSQL	`alerts`, `alert_actions`
Chat Service	MongoDB	`chatMessages`, `channels`, `fileMetadata`
SAR Catalog Service	PostgreSQL+PostGIS	`sar_images`, `sar_metadata`
Agent Service	MongoDB	`agentStates`, `agentConversations`, `planningAids`
Audit Service	MongoDB	`auditEvents`

No service reads from or writes to another service's database. Cross-service data access is via gRPC or RabbitMQ only.

Module Federation Boundaries:

Module	Exposes	Consumes from Core

Module	Exposes	Consumes from Core
Core Shell	Map API, Redux Store, Auth, WebSocket, Logging, Event Bus, Notifications, API Client, Toolbar/Tools/StatusBar registration	—
SA Module	Sites panel, Rules panel, Alerts panel	Map API, Redux, Auth, WebSocket, Toolbar registration
Chat Module	Chat panel, Agent chat	Redux, Auth, WebSocket, Event Bus, Toolbar registration
SAR Module	Catalog browser, Exploitation tools	Map API, Redux, Auth, API Client, Tools registration
Attack Agent Module	Kill chain panels, Planning aids, Traffic-light	Map API, Redux, Auth, WebSocket, Toolbar registration
Admin Module	Config panels, GCS management	Auth, API Client, Toolbar registration

Integration Points & Data Flow

Primary Real-Time Data Flow:

```
External Sensor
    → Sensor Gateway (parse → protobuf)
    → RabbitMQ (sensor.data exchange, routing: {datatype}.{sensor})
    → Detection Service (persist to PostgreSQL+PostGIS, update Redis cache)
    → Rule Engine (evaluate rules, generate alerts → Alert Service)
    → WebSocket Gateway (subscribe RabbitMQ → serialize protobuf → push to
clients)
    → Browser WebSocket Manager (deserialize → dispatch Redux actions)
    → Redux Store → React components + Cesium Entity Layer
```

Kill Chain Data Flow (Attack Agent Demo):

```
Demo Injector → RabbitMQ → Detection Service → WebSocket → Core Shell (map)
                                                              ↓
IronBrain Adapter ← (ATC request) ← Agent Service
IronBrain Adapter → (ATC results) → Detection Service → WebSocket → Core Shell
                                                              ↓
User (via Chat) → Agent Service (Correlation Agent) → Map highlights
                                                          ↓
Agent Service → RoE Adapter → External RoE Evaluator → Alert (HITL gate)
                                                            ↓ (approved)
User creates Planning Aid → Agent Service → MSG Adapter → MSG Weaponeering
```

```
                                                                  ↓
MSG strike plan → Agent Service (Planning Approval Agent) → Traffic-light
assessment
                                                                    ↓ (HITL gate)
Commander approval → Agent Service → Omnisys Adapter → Omnisys tasking
                                                                  ↓
Omnisys BDA plan → Agent Service → Attack Agent Module UI
```

Architecture Validation Results

Coherence Validation

Decision Compatibility: PASS

- All 20+ technology choices verified for version compatibility — no conflicts
- React 19 + Webpack MF 2.0 + Redux Toolkit 2.11 + MUI 7 + CesiumJS 1.138: standard ecosystem
- .NET 10 LTS + xUnit + gRPC + PostgreSQL + RabbitMQ: standard .NET ecosystem
- Node.js 24 LTS + Jest + protobuf + MongoDB + RabbitMQ: standard Node ecosystem
- KeyCloak 26 + Tyk 5.8 LTS + Consul 1.22 + Vault 1.21: proven infrastructure stack
- Jest + ESLint + Prettier + EditorConfig: well-established quality toolchain

Pattern Consistency: PASS

- snake_case (DB/proto) → camelCase (JSON/TS) → PascalCase (C#): consistently applied across all naming tables
- Feature-based frontend organization + layered C# backend: aligned with Module Federation and database-per-service
- RFC 7807 at REST, protobuf on WebSocket/RabbitMQ: clean protocol separation
- Shared ESLint/Prettier configs from ci-templates repo enforce consistency across all 28 repos

Structure Alignment: PASS

- 28 repos map cleanly to the architectural boundaries defined in Steps 4-6
- Each repo independently deployable via its own Helm chart (NFR-M1)
- Tests in `tests/` folder at repo root — consistent across all repo types

Requirements Coverage Validation

Functional Requirements: 87/87 mapped

Domain	FRs	Coverage
Map & Geospatial (FR1-FR15)	15	All mapped to prophesee-core
Detection Management (FR16-FR21)	6	All mapped to prophesee-core + detection-service
Sites Management (FR22-FR26)	5	All mapped to sa-module + sites-service

Domain	FRs	Coverage
Rule Engine & Alerting (FR27-FR34)	8	All mapped to sa-module + rule-engine + alert-service
Collaborative Chat (FR35-FR40)	6	All mapped to chat-module + chat-service + agent-service
AI Agent Kill Chain (FR41-FR54)	14	All mapped across agent-service + 4 adapters + attack-agent-module
SAR Exploitation (FR55-FR60)	6	All mapped to sar-module + sar-catalog-service
Identity & Access (FR61-FR69)	9	All mapped to core + admin-module + KeyCloak + audit-service
Infrastructure (FR70-FR81)	12	All mapped to helm-charts + infrastructure + admin-module
Partner Integration (FR82-FR87)	6	All mapped to 4 adapter repos

Non-Functional Requirements: All categories covered

NFR Category	Covered By
Performance (NFR-P1 to P9)	Redis cache, protobuf WebSocket, entity batching (100ms), Module Federation lazy loading
Security (NFR-S1 to S10)	KeyCloak + Tyk + mTLS + JWT propagation + Vault + audit service
Scalability (NFR-SC1 to SC5)	Helm values per tier (tactical/theater/strategic), stateless services, RabbitMQ, Redis
Reliability (NFR-R1 to R7)	K8s health probes, DLQ retry, circuit breaker, mock/stub adapters
Observability (NFR-O1 to O5)	OpenTelemetry + Prometheus + Grafana + Elasticsearch + structured logging
Maintainability (NFR-M1 to M6)	Independent deployment, buf breaking checks, Helm IaC, Consul, Tyk versioning
Integration (NFR-I1 to I4)	Adapter pattern, timeouts, protobuf validation, independent gateways

Implementation Readiness

Decision Completeness: All critical and important decisions documented with verified versions. Testing (Jest/xUnit), linting (ESLint+Prettier+EditorConfig), CI templates, and documentation structure resolved.

Remaining Deferred (non-blocking):

- MinIO replacement evaluation (RustFS) — post-MVP
- Monorepo tooling for core — evaluate if shared libraries grow
- E2E framework (Playwright vs Cypress) — decide when E2E testing begins

Architecture Completeness Checklist

Requirements Analysis:

- ☑ Project context thoroughly analyzed (87 FRs, 10 NFR categories)
- ☑ Scale and complexity assessed (Enterprise/High)
- ☑ Technical constraints identified (10 constraints)
- ☑ Cross-cutting concerns mapped (8 concerns)

Architectural Decisions:

- ☑ Critical decisions documented with versions
- ☑ Technology stack fully specified (20+ technologies with versions)
- ☑ Integration patterns defined (gRPC, RabbitMQ, WebSocket, REST)
- ☑ Performance considerations addressed (entity batching, protobuf, Redis cache)

Implementation Patterns:

- ☑ Naming conventions established (DB, API, Code, Protobuf)
- ☑ Structure patterns defined (feature-based frontend, layered C# backend)
- ☑ Communication patterns specified (RabbitMQ, Redux, WebSocket)
- ☑ Process patterns documented (error handling, logging, loading states, validation)

Project Structure:

- ☑ Complete directory structure defined (28 repos with full trees)
- ☑ Component boundaries established (Module Federation, database-per-service)
- ☑ Integration points mapped (API, data, module boundaries)
- ☑ Requirements-to-structure mapping complete (498-style traceability)

Quality & DevOps:

- ☑ Testing frameworks decided (Jest, xUnit)
- ☑ Linting/formatting decided (ESLint, Prettier, EditorConfig)
- ☑ CI templates repo defined
- ☑ Documentation repo and structure defined

Architecture Readiness Assessment

Overall Status: READY FOR IMPLEMENTATION

Confidence Level: High

Key Strengths:

- Full FR traceability from requirement to repo/service/database (87/87)
- Clear module boundaries with defined integration contracts
- Comprehensive naming and pattern rules prevent AI agent conflicts
- Multi-repo architecture enables true independent deployment
- Sensor-agnostic gateway pattern proven extensible

Areas for Future Enhancement:

- E2E testing framework selection (Playwright vs Cypress)
- Monorepo tooling evaluation if core shared libraries grow
- MinIO → RustFS migration evaluation
- Performance testing strategy (load testing, Cesium entity benchmarks)

Implementation Handoff

AI Agent Guidelines:

- Follow all architectural decisions exactly as documented in this file
- Use implementation patterns consistently across all components
- Respect project structure and boundaries — do not deviate from directory templates
- Refer to this document for all architectural questions
- Use the Requirements Traceability Matrix to find where each FR lives

First Implementation Priority:

1. `prophesee-data-model` — Define protobuf schemas (this unblocks everything)
2. `prophesee-ci-templates` — Shared configs (ESLint, Prettier, Jest, EditorConfig, pipelines)
3. `prophesee-infrastructure` — K8s, RabbitMQ topology, KeyCloak realm, Tyk, Consul, Vault, Redis, PostgreSQL, MongoDB, MinIO, Prometheus/Grafana
4. `prophesee-core` — Clone WebKit, integrate Module Federation host, Cesium map, Redux store, shared services
5. `prophesee-gateway-demo-injector` — Demo data injection for end-to-end validation
6. `prophesee-websocket-gateway` — Real-time push from RabbitMQ to browser
7. First backend services: detection-service, alert-service, rule-engine
8. Frontend modules: sa-module, then chat-module, attack-agent-module